# FINAL REPORT / A005
for
Contract # DASG60-95-C-0103
Distributed Simulation of Synthetic Environments and Wireless Networks

August 1, 1995 through July 31, 1999

Principal Investigator: Richard M. Fujimoto

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
(404) 853-9384 (Fujimoto)

September 3, 1999

19990930 025

Contents

Chapter 1

Executive Summary

The goal of this research is to provide a decision aid that allows engineers, scientists and battle-managers to simulate complex situations and facilitate rapid and informed planning. Specifically, we are developing mechanisms to facilitate the decision process in battle management. Often, very large systems may need to be modeled such as an entire combat theater or a large airspace. The problem with large systems is that a single simulation run may take hours to complete. In order to reduce execution time, one can decompose a sequential simulation into a collection of many interacting programs distributed over multiple processors. This approach is called parallel simulation. To ensure correctness, we require that parallel execution generates the same result as a sequential execution. This technology enables off-line decision aid simulations to be transformed into real-time decision aids for time critical situations without loss of model accuracy.

To build fast, efficient and powerful simulators we need parallel algorithms that

1. hide computational latencies for image generation in virtual environments,
2. allow effective what-if and alternative scenario analysis and decision making in time critical situations,
3. maximize forward progress via background execution,
4. implement interactive visualization techniques that provide insight into parallel simulation execution to prevent potential bottlenecks and
5. provide for geographically distributed (WAN) environments and exploit the capabilities of high performance interconnections to deliver real-time results.

To address these issues the development and integration of new mechanisms for interactive parallel discrete event simulations are presented, implemented and evaluated.

1.1 Contributions

The motivation is to demonstrate speed-up in real-word simulations via optimistic techniques. The focus is on the development of mechanisms to facilitate the decision process in battle management. An interactive decision tool leveraging new techniques in parallel discrete event simulation is being designed and implemented for this project. The principal technical contributions of the work performed under this contract are summarized below.

1.1.1 Real-Time Simulation of Large Scale Battle-Management Systems

A portion of a large-scale battle-management system, the THAAD Integrated System Effectiveness Simulation (TISES), is parallelized as proof-of-concept using new and existing optimistic computation techniques. TISES is a high-fidelity simulation that models all activities performed by a collection of THAAD missile batteries during an engagement scenario.

The TISES source includes approximately 300,000 lines of code. Because of its large size and complexity, we use an incremental approach where piece by piece the entire system is parallelized via a spiral software development process. In this research the eye of the spiral is the Threat Evaluation and Weapon Assignment (TEWA) module of the BMC^3I component which is central in the decision process. Our performance results and evaluation show that optimistic techniques provide a viable approach to reducing the execution time of BMC^3I applications. TISES is situated in an interactive environment developed for parallel simulators that allows scenario analysis using monitoring and potential steering capabilities. These mechanisms serve as an enabling technology for the development of faster-than-real-time BMC^3I applications and provide battle commanders a "hot-situation" analysis tool and decision aide.

1.1.2 Optimistic Computations in Virtual Environments

An optimistic I/O mechanism is introduced to provide for reduction of computational latencies (as in image rendering for virtual environments). Using this mechanism virtual environments can exploit optimistic Time Warp protocols to request images before they are needed. A roll-back computation enables recovery whenever an unexpected real-time input forces the cancellation of scheduled operations. Evaluation on a shared memory multiprocessor demonstrates how virtual environments can benefit from optimistic computing.

1.1.3 Cloning Parallel Simulations

A parallel simulation cloning algorithm that supports rapid evaluation of several possible alternative futures is introduced and developed for this project. Cloning supports decision making tasks such as determining whether or not to mobilize reinforcements in a military scenario. Other applications that may benefit include gaming, strategic and tactical battle planning and air-traffic management.

This report includes a comprehensive performance evaluation of "cloning" parallel applications using Clone-Sim, a software library. Clone-Sim is developed as part of this research and provides for interactive multi-scenario analysis. Currently, the library is integrated with our optimistic simulator Georgia Tech Time Warp (GTW), however it is independent of whether the underlying simulation model is optimistic or conservative.

1.1.4 Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms

For Time Warp programs executing on a NOW environment, there are internal and external workload sources that must be taken into consideration if efficient execution is to be maintained. The principal contribution of this work is devising a single algorithm that is able to mitigate both kinds of irregular workloads. The observation driving this algorithm is that in order for a Time Warp program to be balanced, the amount of wall-clock time necessary to advance an LP one unit of simulation time should be about the same for all LPs in the system. In particular, we demonstrate using a PCS simulation model as well as a synthetic application that our Background Execution Algorithm (BGE) is able to:

- dynamically allocate additional CPUs during the execution of the distributed simulation as they become available and migrate portions of the distributed simulation workload onto these machines,
- dynamically release certain CPUs during the simulation as they become loaded with other, external, computations, and off-load the workload to the remaining CPUs used by the distributed simulation, and
- dynamically re-distribute the workload on the existing set of processors as some become more heavily or lightly loaded by changing, externally or internally induced workloads.

1.1.5 Visualizing Parallel Simulations

While network computing environments are evolving into a popular and effective platform for optimistic parallel discrete-event simulation systems, they are not without difficulties. These environments are subject to uncontrollable external loads and typically composed of workstations of varied computational capabilities. These factors can greatly degrade the performance of parallel discrete event systems (PDES). This work investigates the use of graphical visualization to provide insight into the execution of a simulator developed for network computing environments. Specifically, our investigation is focused on the augmentation of a general-purpose network computing visualization system with middleware-specific visualizations and has resulted in the PVaniM-GTW visualization system. The synergies provided by the general-purpose graphical views combined with the newly developed middleware-specific views enable us to provide more insight into PDES middleware than a generic network computing visualization system. Moreover, the end-user is able to utilize the new insights provided by the system for any simulation application without the expense of developing his or her own custom application-specific visualizations for purposes such as performance analysis and understanding simulator execution. We believe systems such as PVaniM-GTW that employ middleware-specific views provide a cost-effective compromise between both extremes of the visualization tool spectrum. In our case study, PVaniM-GTW enable us to identify performance characteristics of the GTW system and weaknesses in the BGE algorithm.

In the future we plan to correct the performance anomalies discovered using PVaniM-GTW and evolve the visualization system to better support clusters of uniprocessor and multiprocessor machines. We also intend to provide collaborative visualization support for GTW simulations.

1.1.6 RTI Performance on Shared Memory and Message Passing Architectures

The U.S. Department of Defense (DoD) mandated that the High Level Architecture (HLA) be the standard architecture for DoD modeling and simulation programs [DMSO 1998]. This requires that the HLA span a wide range of applications, computing architectures, and communication paradigms.

Previously most implementations of the High Level Architecture (HLA) Run Time Infrastructures (RTIs) have been implemented on a Network of Workstations (NOWs). Typically these workstations are connected via TCP/IP networks or other high speed interconnects such as Myricomm's Myrinet [The Myrinet 1998].

Recently Shared Memory Multiprocessors (SMPs) have been considered as a platform for HLA RTIs. SMPs offer a different communication paradigm. An HLA RTI implemented over a NOW must use message passing to transfer information between workstations, e.g., using UDP or TCP packets over a TCP/IP network or Fast Messages (FM) [Pakin et al.] over a Myrinet network. In SMPs communication between processors is realized through the use of shared memory. For example, to transfer information from one processor to a second, the first processor writes to a memory location in shared memory and signals the second processor to read from the same location.

Under this contract we have developed a shared memory time management algorithm. This algorithm has been implemented as an RTI-Kit [Fujimoto and Hoare 1998] library and its performance is compared to an existing message passing time management algorithm. RTI-Kit is designed to enhance performance and add new functionality of existing RTIs by exploiting capabilities of a high performance interconnect.

A goal of this work is to federate Georgia Tech Time Warp (GTW) simulations using RTI-Kit. To facilitate this effort and to exploit trade-offs between performance and network properties we are evaluating different communication architectures for networked symmetric multi-processors.

Chapter 2

Introduction

Fast, efficient and interactive simulation environments can be constructed using new techniques developed for parallel discrete event simulations. These environments allow engineers, scientists and system administrators to investigate complex situations and/or facilitate rapid and informed decision making.

As an example of how interactive simulation can be used, consider battle-management. Here, the controller could benefit from an on-line interactive simulation to determine which strategy is most effective, such as the firing of an anti-ballistic missile. If an unexpected event occurs such as the presence of incoming threat missiles, the controller interacting with the simulation may want to evaluate the effect of activating different tactical strategies (such as fire missile or don't fire). This can be done by simulating each of the strategies and then evaluating, comparing and monitoring each alternative to determine which strategy offers the most effective solution. Inefficiencies in the battle management system can be discovered by monitoring the simulation then allowing backward execution to evaluate cause and effect relationships. Currently available simulation tools often cannot provide this capability, because simulation runs may take hours to complete while decisions must be made in minutes.

Interactive simulation also serves a valuable role in training tasks. Fighter pilots routinely utilize interactive virtual world simulations to fine-tune combat skills. Networked simulators allow pilots from many different geographical locations to participate in the same distributed exercise. A virtual world can offer advantages that are not available in a real world training exercise such as immediate replay of a scenario to analyze trainees' performance. Further, such simulations are (1) less expensive (2) safer and (3) more environment friendly than real-world exercises.

The battle-management example requires a faster-than-real-time analytical simulator where the goal is to capture anticipated system behavior as rapidly as possible. The virtual environment (fighter combat) example requires a real-time simulator with users potentially at geographically different locations. Both of these applications motivate the use of multiple computers to execute the simulation. In the first case, multiple processors are used to reduce the execution time of the simulation program. In the second they are required to support the computational demands of large virtual environments and facilitate linking existing simulation programs developed for different platforms.

The next sections provide a context for this work through a brief overview of different classes of computer simulations, including sequential, parallel and distributed approaches. The discussion also distinguishes between conservative and optimistic synchronization techniques. The chapter also gives an overview of our simulation executive the Georgia Tech Time Warp kernel (GTW) that was utilized for this research.

2.1 Discrete Event Simulation

In this work a "simulation" is a computer program that models the behavior of some actual or envisioned system over time. A simulation computes possible behaviors of the environment to aid in evaluating potential outcomes and/or aid on-line decision making.

A computer simulation program contains a collection of state variables characterizing a model of the environment, a set of behaviors or functions that effect changes in the variables and time management mechanisms that allow the simulation to progress or evolve through simulation time. Simulation time is an abstraction of physical time, where simulated time defines the order of events in the simulation.

There are two major classes of simulations differing in their treatment of simulation time: continuous and discrete. This research is concerned with discrete event simulations where changes in state variables are viewed as occurring at discrete points in simulated time. In contrast, continuous simulations view state changes as occurring continuously over time. In continuous simulations changes are typically described by differential equations. Applications usually simulated by continuous approaches include the modeling of weather, surface transformations and viscous fluids. Typical discrete event simulation applications include modeling of air-traffic, battle-management and communication networks.

Discrete event simulations typically maintain data structures of state variables, an event list of forthcoming time-stamped events and a global clock variable which denotes progress in simulated time. Events define system-state changes and model interesting actions in the system (such as an arrival event or a departure event in an airport simulation). The simulation progresses by repeatedly processing the event containing the smallest time stamp from the event list. The current event may effect the state or may schedule new events that are inserted into the event list. When an event is processed the global clock is updated to the time stamp of the event. For example, a simulation may model an air-traffic system where state variables indicate the number of airplanes at each airport. Departure and arrival events modify these variables as new aircrafts arrive or depart from the airport.

2.2 Parallel Discrete Event Simulation

Often, very large systems may need to be modeled such as an entire combat theater or a large airspace. The problem with large systems is that a single simulation run may take hours to complete. In order to reduce execution time, one can decompose a sequential simulation into a collection of many interacting programs distributed over multiple processors. A *correct* parallel simulator generates the same result as a sequential execution.

In this work, a parallel simulation is composed of distinct components called logical processes or LPs. Each LP models some part of the simulation. For example in a battle-management system a Battalion Level Tactical Operations Center (BTN-TOC), a Battery Level Tactical Operations Center (BTY-TOC) and a Launcher may each be represented by an LP, as shown in Figure 2.1. The logical processes are often mapped to different processors and communicate by exchanging time-stamped event messages. As in a sequential simulation a change in system state is defined

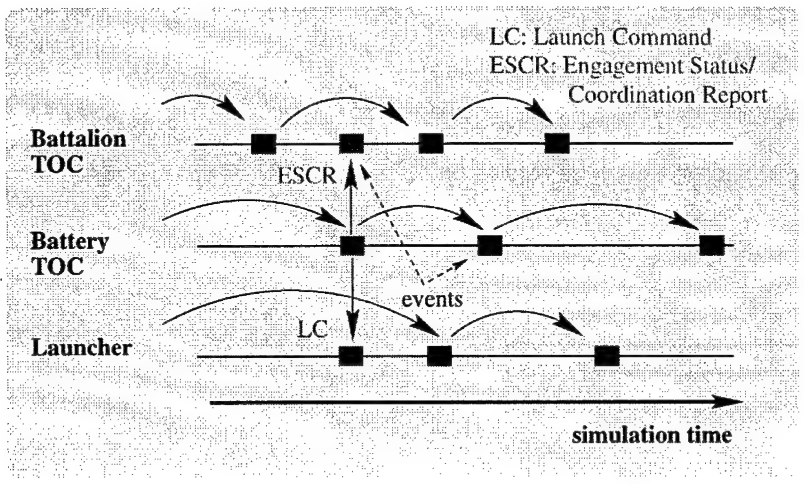


Figure 2.1: A Snapshot of a Battery-Management System: A Battery Level Tactical Operations Center transmits a Launch Command message to a selected Launcher and simultaneously sends Engagement Status/Coordination Report (ESCR) messages to each adjacent Tactical Operations Center.

by an event. The "scheduling" of an event is accomplish by sending a message that may request the destination LP to change its state or schedule additional events. For example a launch event may be scheduled when a launcher command is processed at a Battery Level Tactical Operations Center. To summarize: distinct components in the simulation are modeled by logical processes and the simulation progresses as LPs exchange time-stamped event messages that cause changes in the system state at discrete points in time.

A synchronization mechanism is required to ensure that each LP processes events in time-stamp order. The two prevailing classes of synchronization protocols are called *conservative* and *optimistic* mechanisms. The conservative protocol enforces consistency by avoiding the possibility of an LP ever receiving an event from its past (as measured in simulated time). LPs *wait* to process events until reception of an out of order event is impossible. The optimistic protocol, in contrast, uses a detect-and-recover scheme. When an event is received in the past of an LP it recovers by rolling back previously processed events with later time-stamps than the one that was just received.

2.2.1 Conservative Protocols

Conservative protocols typically use the notion of a link. A link is a communication path between two LPs in the simulation. Early protocols assumed (1) that links are defined at the inception of the simulation, (2) time stamps of messages delivered on each link form a non-decreasing sequence and (3) communications are reliable. From this, one can conclude that the last message received on a link is a lower bound on all future events that will be later received on that link.

In order to avoid receiving an event in its past an LP keeps track of the smallest unprocessed time stamp of any message buffered on each of its incoming links, or if there are none, the time stamp of the last message received on the link. This is accomplished by associating a variable called the *link time* to each of its links. The algorithm proceeds by processing the event associated with the smallest link time. But if there are no events associated with a specific link and the current link time is the smallest, the LP must wait until it receives a new message over that link. Otherwise this link may produce an event with a smaller time-stamp than events it received on its other links.

Since conservative mechanisms wait or block until no events can be received from the past they are prone to deadlock: a cycle of empty links may arise if each LP waits on the next LP within the cycle. Conservative protocols must avoid or detect and recover from deadlocks.

Early mechanisms suggested independently by [Bryant 1977] and [Chandy and Misra 1979] avoid deadlock situations using "null-messages". A null message is sent by an LP to all its neighboring LPs after processing every event. This allows the receiver to define a new lower bound on its link. Null messages insure that a logical process will not send new messages with a lower time stamp. As null messages advance link times, the lowest time-stamped event in the system will eventually be ready for processing, thus avoiding deadlock. In order to reduce the number of null messages protocols described in [Peacock et al. 1979] and [Nicol and Reynolds, Jr. 1984] only send null messages when an LP requests one. An LP requests null message when it realizes a link is empty and before it blocks.

Another approach by [Chandy and Misra 1981] allows deadlocks but uses a detection and deadlock breaking mechanism to recover. The approach by [Lubachevsky 1989] reduces the search space for safe events by using time windows. Only events with time-stamps within the time window are eligible for processing. Finally, synchronous approaches process safe events in iterative global phases using barrier synchronizations.

2.2.2 Optimistic Protocols

The prevalent optimistic approach utilizes the Time Warp mechanism originally developed by Jefferson and Sowizral [Jefferson and Sowizral 1982]. LPs process incoming messages optimistically without blocking. Time Warp is consider to be "optimistic" because it allows LPs to speculatively process events and only synchronizes LPs when events are processed out of time-stamp order. This protocol has demonstrated some success in speeding up a variety of simulation applications, including combat models [Wieland et al. 1989], communication networks [Presley et al. 1989], wireless networks [Carothers et al. 1995], queuing networks [Fujimoto 1989a], and digital logic circuits [Briner 1991], among others.

The Time Warp synchronization protocol can be divided into two parts: a local control mechanism and a global control mechanism, as show in Figure 2.2. The *local control* mechanism detects any out-of-order event computations. When such an error occurs at an LP, that LP must be "rolled back" to an event time that is just prior to the causality error. For example, as depicted in Figure 2.2 a simulation of a battle-management system consisting of three modules: A Battalion Level Tactical Operations Center (BTN-TOC), a subordinate Battery Level Tactical Operations Centers (BTY-TOCs) and a launcher are each assigned to are each assigned to a separate logical process and processor. Consider the case where the Battalion TOC schedules a new Area of Responsibility (AOR) message in the "past" of the Battery TOC LP. Here, the Batter TOC LP must account for the AOR message before resuming. This is accomplished by a two step process. First, any event computations time-stamped at a later time than the AOR message of the Battery TOC LP's state must be undone. Time Warp accomplishes this by check-pointing the state of the LP prior to executing the event. Second, any future events that were scheduled as part of an incorrect event computation are retracted. Consequently, cascaded rollbacks are possible, thus making it imperative that incorrect event computations be erased as soon as possible before it spreads throughout the entire simulation. To avoid these cascades Time Warp can use aggressive cancellation, that

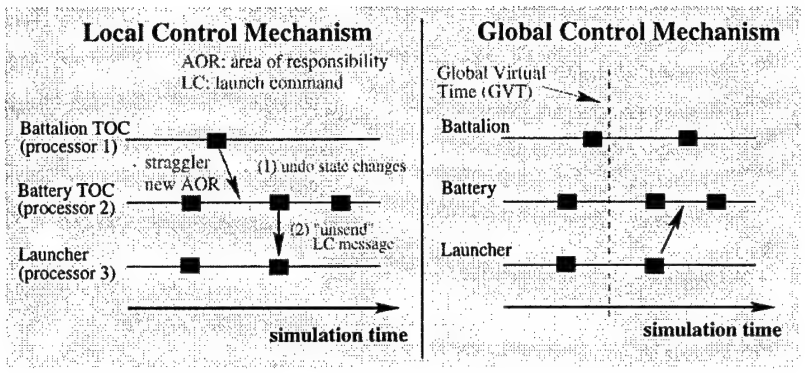


Figure 2.2: Time Warp Synchronization Mechanism.

immediately retracts messages.

Because changes made to an LP's state are recorded to enable roll-backs, a computer's memory would be exhausted shortly without some mechanism to reclaim old versions of state, and processed events. This action is performed by the *global control* mechanism. Here, a lower bound on the time-stamp of any future rollback is determined. This time is called Global Virtual Time (GVT). Any recorded state changes or processed events that are older than GVT may be reclaimed for future use. This process is called fossil collection. Additionally, irrevocable operations such as I/O to disks or a display can be performed when GVT sweeps past.

The optimistic approach was shown by Lipton and Mizel [Lipton and Mizell 1990] to outperform Chandy-Misra by a factor of n using n processes in certain cases, but Chandy-Misra can only outperform Time Warp by at most a constant factor when roll-back cost is constant. A reason for this is that LPs can exploit parallelism by executing independently without waiting and without any global control (except GVT). In effect, Time Warp algorithms have a greater potential to take advantage of parallelism than conservative protocols.

2.3 Georgia Tech Time Warp (GTW)

If one were to naively implement the Time Warp algorithm, very little if any increase in performance would be realized. For the past decade, researchers at Georgia Tech have been developing new algorithms and techniques that are necessary to make Time Warp a viable technology. This research has resulted in a highly optimized parallel simulation executive called *Georgia Tech Time Warp (GTW)*.

As new algorithms and techniques are developed, they are incorporated into GTW. Representative results from our research include:

- Analytic models that aide to improve our understanding of Time Warp performance [Gupta et al. 1991],
- Adaptive memory-based throttling [Das and Fujimoto 1994] and fast error correction that prevents rollback thrashing [Fujimoto 1989a],
- Fast GVT algorithm for shared memory multi-processors [Fujimoto and Hybinette 1997], and

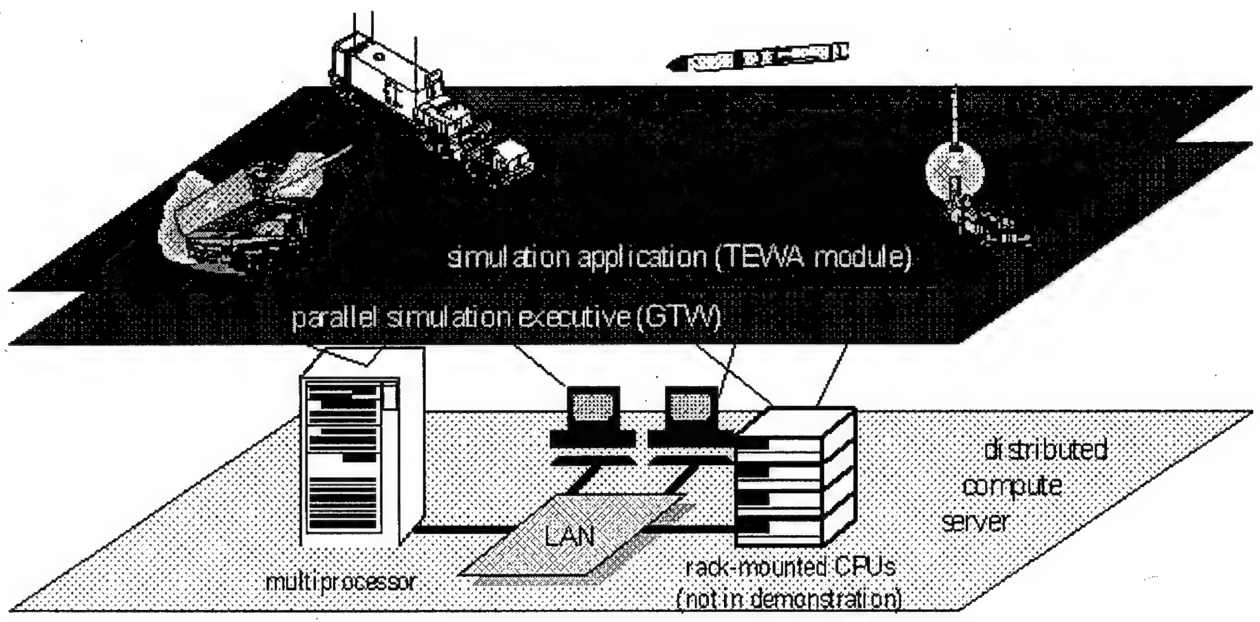

simulation application (TEWA module)
parallel simulation executive (GTW)
multiprocessor
LAN
rack-mounted CPUs (not in demonstration)
distributed compute server

Figure 2.3: System Architecture.

- Fast, incremental on-the-fly fossil collection [Fujimoto and Hybinette 1997]

The system architecture for GTW is shown in Figure 2.3. GTW provides an API that insulates the application from the underlying hardware platforms. Currently, GTW executes on a combination of shared-memory multiprocessors and heterogeneous workstations.

2.4 Virtual Environments

In contrast to analytic simulations, virtual environments include either human interactions or hardware realizations as an integral part of the simulated system. Virtual worlds require real-time execution as opposed to analytic simulations that often proceed faster than real-time. Virtual environment simulations often exploit limitations of human perception. For example, events happening close together in time can often be processed in any order since a human cannot perceive the actual ordering.

Development of distributed virtual world simulation was initiated by the military to reduce the cost of training and to improve safety in the early 1980's. The SImulator NETworking project or SIMNET demonstrated the feasibility of distributed virtual environments [Kanarick 1991]. SIMNET connected over 200 simulators in 7 remote locations in 1990.

The success of SIMNET lead to the development of a standard protocol called Distributed Interactive Simulation (DIS). DIS differs from SIMNET in that it allows the connection of heterogeneous simulators. A key goal of DIS is to support interoperability among different simulators, systems and embedded devices as well as human participants.

Independent of DIS, the Aggregated Level Simulation Protocol (ALSP) was initiated in 1990 to extend SIMNET to the analysis of war-gaming [Wilson and Weatherly 1994]. War-gaming simulations are sometime called constructive simulations, because they model components at a low level of detail (such as a battalion instead of an individual tank or soldier). In contrast to DIS, ALSP processes events in strict time-stamp order. This is accomplished using the Chandy/Misra/Bryant null message protocol described earlier.

To merge ALSP and DIS technologies the development of the High-Level Architecture (HLA) began in 1993. HLA bridges the gap between analytical and virtual environment and standardizes the development of modeling and simulation activities within the department of defense. A key component of this integration is time management that includes a mechanism to ensure event ordering when needed [Fujimoto and Weatherly 1997].

2.5 Overview of the Report

The remainder of the report is organized as follows: Chapter 3 gives details of the parallelization of the battle-management application: TISES. Chapter 4 which describes a new mechanism called optimistic I/O that speedup computationally intensive image generation in virtual environments. The next two chapters: 5 and 6 discusses "cloning" of parallel applications. The former chapter discusses Clone-Sim, the software library implemented and developed, its usage and programmer application interface. The latter gives details of the cloning algorithm and its performance. Dynamic load balancing on cluster computing environments is discussed in Chapter 7. Chapter 8 discusses the evaluation and implementation of a general purpose network computing visualization system, called PVaniM, and Chapter 9 reports on the implementation of RTI-Kit on SGI and over TCP, enabling its use in geographically distributed (WAN) environments. This chapter also include a performance evaluation of different communication architectures for networked symmetric multi-processors in order to exploit the trade-offs between performance and network properties.

Chapter 3

Real-Time Simulation of Large Scale Battle-Management Systems

The driving motivation of the research described in this chapter is to provide a decision aid that allows engineers, scientists and battle-managers to simulate complex situations and facilitate rapid and informed planning. In this chapter, we are describing mechanisms developed to facilitate the decision process in battle management.

Often, very large systems may need to be modeled such as an entire combat theater or a large airspace. The problem with large systems is that a single simulation run may take hours to complete. In order to reduce execution time, one can decompose a sequential simulation into a collection of many interacting programs distributed over multiple processors. This approach is called parallel simulation. A *correct* parallel simulator generates the same result as a sequential execution. This technology enables off-line decision aid simulations to be transformed into real-time decision aids for time critical situations without loss of model accuracy.

A portion of a large-scale battle-management system, the THAAD Integrated System Effectiveness Simulation (TISES), is parallelized as proof-of-concept using optimistic computation techniques. The primary technical objectives of this work are:

- to understand the technical issues in applying optimistic parallel simulation technology to a real-world battle management simulation system (BMC^3I in particular) and develop solutions,
- to develop a practical methodology to parallelizing an existing, large-scale Ada sequential simulation,
- to demonstrate the technical feasibility of this approach by implementing a scaled-down parallel version of the simulator, and
- to evaluate the increase in simulator performance that can be obtained by exploiting optimistic parallel simulation technology.

3.1 Overview of TISES

The objective for the TISES system is to achieve and end-to-end system simulation of a THAAD multi-battery configuration. The output data produced by TISES for a particular scenario includes

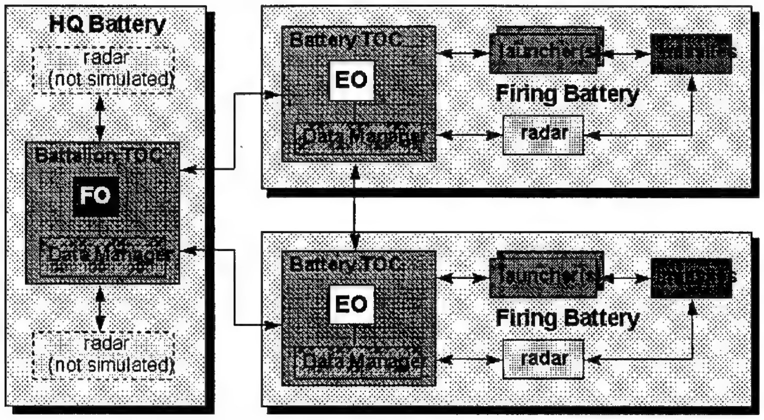


Figure 3.1: TISES Simulated THAAD Communications Architecture.

the following measures of effectiveness:

- system and timeliness,
- radar and loading performance,
- communication network performance,
- BMC^3I performance, and
- missile performance.

As depicted in Figure 3.1, a multi-battery THAAD system consists of a Battalion Level Tactical Operations Center (BTN-TOC) and a number of subordinate Battery Level Tactical Operations Centers (BTY-TOCs). Each BTN-TOC contains a number of co-located radars, which are not simulated by TISES, as well as communication links to each subordinate BTY-TOC. A BTY-TOC contains three to nine local launchers with up to fifteen missiles each, one local radar, and communication links back to the BTN-TOC plus communications links between BTY-TOCs.

3.2 Sequential Implementation

In terms of software implementation, TISES is a non real-time discrete event, engineering design-level simulation of the THAAD system. While not completely object-oriented because it is written in Ada, this system was designed to be highly modular. Figure 3.2 shows the functional decomposition of the TISES system. This system is decomposed into nine sub-components. Each sub-component is realized as an Ada-task. Controlling which task executes is the responsibility of the simulation Auto Driver. This task peers into the global event list and selects the event with the smallest time-stamp and removes it from the list. Next, the Auto Driver will examine which sub-component is responsible for processing this particular event type and proceeds to schedule the Ada-task modeling that sub-component, passing the event as an argument. When a sub-component task is given control, it invokes its generic event handler and based upon the event type determines the appropriate event

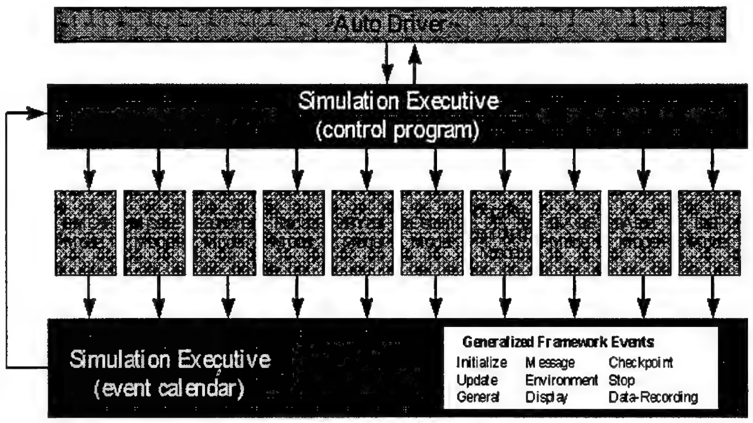


Figure 3.2: TISES Simulation Executive (sequential).

processing procedure to invoke. This procedure will execute the necessary steps to complete the event. Should it be necessary, an event processing procedure may schedule new events into the future, which are posted to the global event list. Once the procedure is complete, the generic event handler will return control to the Auto Driver, which in-turn will select the next event to be processed.

The TISES system, due to its high-fidelity, is a large system simulation consisting of about 300,000 line of code, mostly in Ada, however, some components are written C and Fortran. Because of the size of this simulation, parallelization of the entire TISES system was well beyond the scope of our proof-of-concept effort. Consequently, we narrowed our parallelization effort to the Threat Evaluation and Weapon Assignment (TEWA) module of BMC^3I sub-component. In the following sections, we will first provide a high-level overview of the BMC^3I sub-component and then discuss the TEWA module.

3.2.1 BMC^3I Sub-component

The goal of the TISES BMC^3I sub-component is to accurately simulate the functionality associated with the THAAD tactical operations battle management, command, control, communications, and intelligence behavior. Simulation of TOC processes is the primary focus of this sub-component. These processes can be divided into the following five functional areas:

- *Force operations:* each battery TOC is sent a set of areas of responsibility and rules of engagement by the battalion TOC.
- *Sensor management:* generates and sends search/stop commands and search cue commands.
- *Track management:* receives object and discrimination reports as well as distributes track update reports.
- *Threat evaluation and weapons assignment:* allocates assets to targets, estimates asset damage, generates and selects engagements.
- *Engagement control:* monitors remote and local engagements.

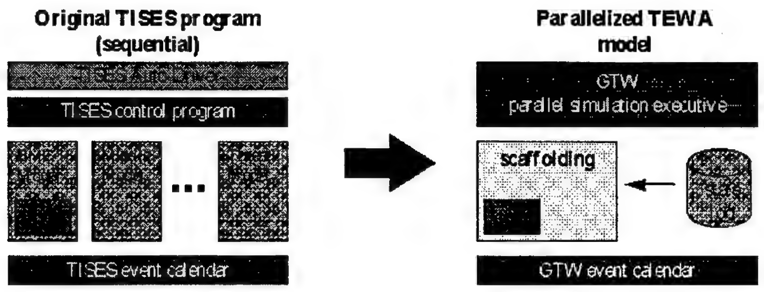


Figure 3.3: Incremental Parallelization Methodology.

3.2.2 TEWA Module

The Threat Evaluation and Weapon Assignment module utilizes sophisticated look-ahead planning techniques. This module consists of the following three major parts:

- *Threat Evaluation:* associates targets to assets and estimates asset damage using a greedy optimization strategy that is based upon a likelihood function.
- *Generate Engagements:* a set of candidate engagements is produced using an interpolation method that individually meet all timing and geometry constraints.
- *Select Engagements:* selects the set of coordinated engagements which satisfies the rules of engagement within the collective timing and resource constraints.

Once a threat has been evaluated and an acceptable engagement has been determined, the TOC will transmit a Launch Command message to the selected launcher and a Interceptor Support Plan message to the supporting radar. At the same time the Launch Command message is being sent and Engagement Status / Coordination Report (ESCR) messages are sent to each adjacent TOC to denote previously unreported changes in engagement status. These ESCR reports will play a pivotal role in determining the performance of the parallelized TEWA module. This aspect will be discussed in future sections.

3.3 Parallel Implementation

The TISES software was parallelized using an incremental parallelization methodology. Here, there is a spiral software development process where piece by piece the system is parallelized. As a new piece is added to the parallelized version, it can be tested for correct functionality, thus a working parallel version of the system is always maintained.

The specific implementation of our incremental approach is shown in Figure 3.3. First, we extracted the TEWA module from the BMC^3I sub-component of TISES. To reproduce the behavior of the non-parallelized portions a scaffolding layer around the TEWA module was build. The scaffolding layer was implemented by first instrument the BMC^3I sub-component to capture snapshots of the BMC^3I system state prior to processing each TEWA event and write them to log files located on local disk. The TEWA/GTW system can then dynamically load the appropriate state snapshots

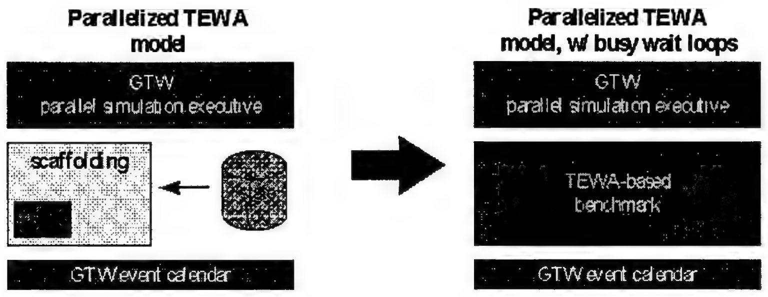

Parallelized TEWA model
GTW parallel simulation executive
scaffolding
GTW event calendar
Parallelized TEWA model, w/ busy wait loops
GTW parallel simulation executive
TEWA-based benchmark
GTW event calendar

Figure 3.4: Performance TEWA benchmark program from parallelized TEWA model.

from the log files prior to processing each TEWA event, giving the parallelized TEWA module the illusion that all of TISES is running.

In order to realize a parallelized TEWA module, we needed to determine how best to map TISES simulation objects to GTW's underlying simulation paradigm, which consists of LPs and events that travel between LPs. How this mapping is realized can be a critical factor in determining parallel system performance. The key lies in finding simulation objects that communicate or send messages infrequently, since more remote or "off-processor" messages increase the likelihood an LP will roll back. In the TEWA module, TOC objects were chosen to serve as LPs since they only communicate when a missile is launched. Recall, that after a missile launch, a TOC will inform all other TOCs by sending an ESCR message.

A second program that precisely models the computational requirements of the real TEWA module without the perturbation of the scaffolding, as shown in Figure 3.4 was developed. The performance of TISES can thus be assessed more realistically. In this TEWA-based benchmark program, busy wait loops are used to reproduce the event computation time of each TEWA event. A single TEWA event may take between 50 milli-seconds to 6 seconds to execute, depending on the number of threats a TOC needs to consider.

3.4 Performance Results

Using the benchmark program, we conducted a series of experiments designed to determine the performance of the parallelized TEWA/GTW system. For all experiments, the China-Taipei scenario is used. This scenario generates 57 threat objects, of which 36 are determined to be lethal. Additionally, this benchmark program includes all Time Warp specific state saving and parallel execution overheads (i.e., rollbacks, message cancelations, etc.). In the first experiment, a 64 TOC configuration is used and the speedup as a function of the number of processor used on a fixed problem size is calculated. The sequential TEWA module's execution time was empirically determined to be 259 seconds. As shown in Figure 3.5, we observe that the TEWA/GTW system obtains good speedups for all numbers of processors (up to 12 on 16 processors). Efficiency is defined to be the percentage of computation that was not rolled back. Here, we clearly see high efficiencies for all processor configurations, which is due to the high degree of parallelism that is available in the TEWA module.

In the second set of experiments, speedup is calculated for a scaled problem size as a function of

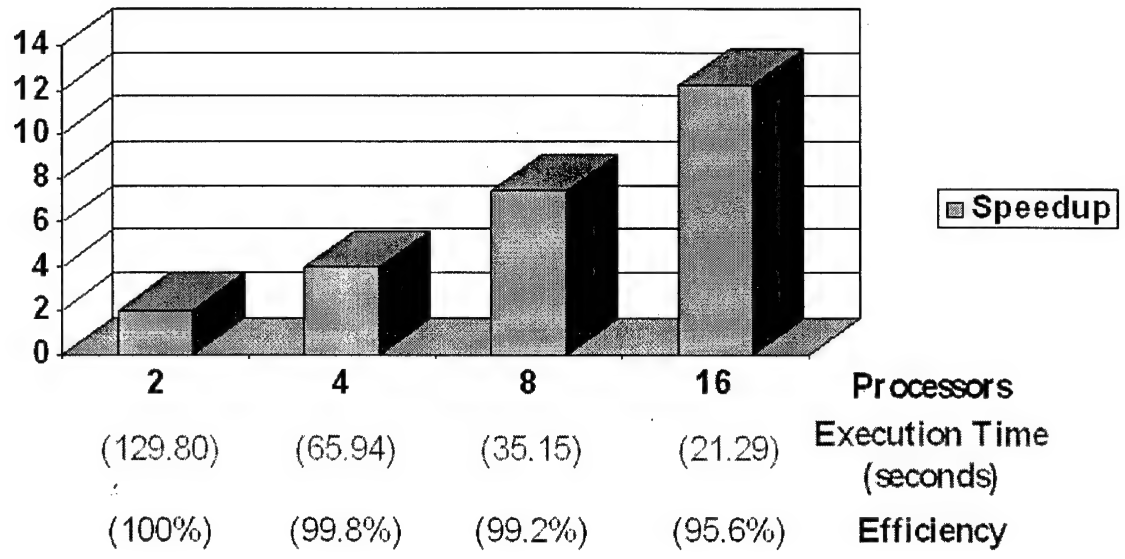


Figure 3.5: Performance measurements across varying number of processors for a fixed problem size.

the number of processors used. In scaling the problem size, we used 4 TOCs as our scaling factor. By that we mean that each processor has 4 TOC LPs mapped to it. For example, the 2 processor case models 8 TOC LPs, the 4 processor case models 16 TOC LPs and so on. The results from these experiments are shown in Figure 3.6. Like the previous set of experiments, we observed high efficiencies and good speedup across all processor configurations, which is again attributed to the high degree of parallelism that is available within the TEWA module.

3.5 Interactive TISES

TISES can benefit from new research in interactive parallel discrete event simulation. The benefit is not only from the speedup of parallelization, also insight from monitoring and potential steering capabilities can be valuable. Currently, TISES produces output to a file on disk. The same file can also be used after the parallelization of TISES to monitor performance metrics. Information from the file is to be extracted by a separate and independent process that channels pertinent information to a graphical user interface written in Java.

The software architecture is shown in Figure 3.5. The TISES modules, shown as LPs in Figure 3.5, are parallelized within the GTW executable. The modules are scheduled by the GTW kernel. The modules produces a file that is read by a separate process (PE) that channels information via an output string to a graphical user interface. As mentioned above TISES is situated in an interactive parallel discrete event programming environment which is describe in more detail in the section below.

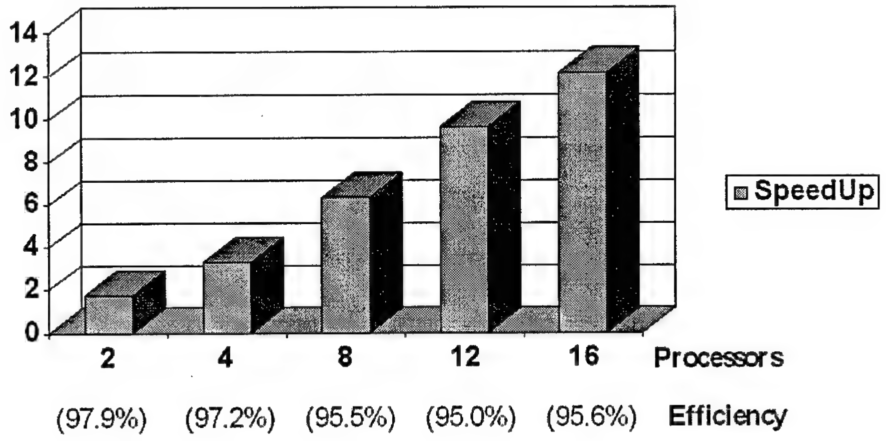


Figure 3.6: Performance measurements across varying number of processors for a scaled problem size.

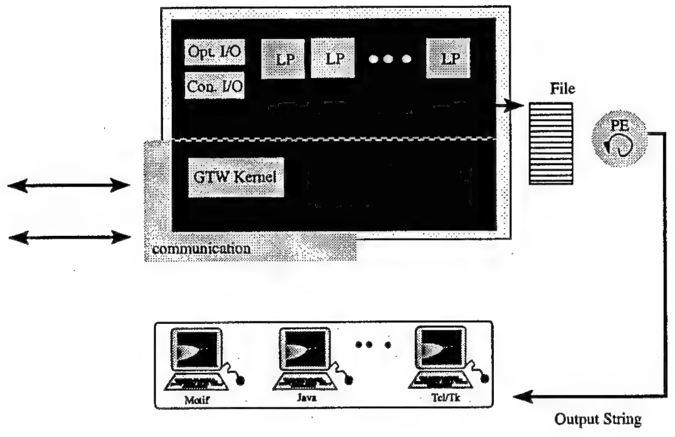


Figure 3.7: GTW and TISES Architecture

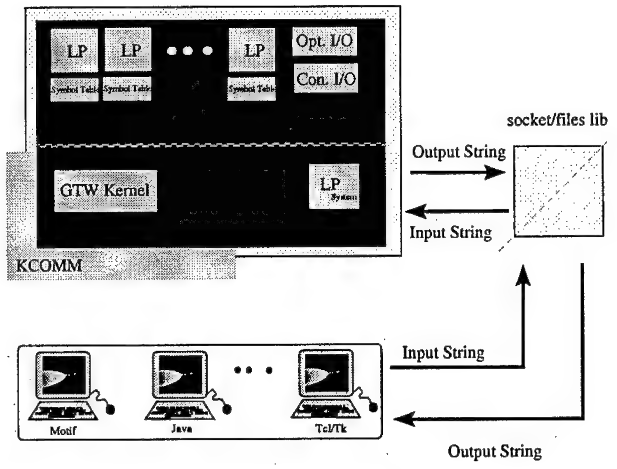
3.6 Interactive Parallel Discrete Event Simulation

The goal for the interactive parallel discrete event system (IPDES) is to provide a decision tool that allows researchers to accurately simulate complex situations and facilitates quick and informed decisions.

In order to deliver a powerful decision tool, our design provides for pausing, steering, monitoring, rollback, reverse execution and the evaluation of alternatives within the simulation (a generalization of what-if scenario analysis). Pausing the simulation is important when a participant wishes to interject immediate or future branches into the simulation as *decision points*. Steering is the insertion of decision points without pausing. Steering and pausing can also provide for modification of simulation parameters and variables. Monitoring is a process whereby specified variables in the simulation are sampled. The variables may be sampled continuously or triggered when a predefined condition occurs, such as the expiration of an execution path. Rollback provides for the re-evaluation of old decision points and variables. Reverse execution enables the system to re-construct the events leading to an outcome. While the design support all these features, the below focuses on the details of the graphical user interface that provides for the parallel evaluation of TISES. A later chapter introduces a new interactive mechanism for the exploration of what-if and alternative scenarios.

The objectives are to provide this functionality transparently to the application program that simulates the target domain and to do this efficiently. Transparency means that the simulation programmer does not need to make explicit calls to monitor or steer simulation variables. This can be done by pre-processing the simulation software's symbol table to *generate* code that supports the monitoring. A hash table can be used to enable efficient monitoring and steering of variables on demand. The table is distributed among logical processes to enable fast lookup.

The interactive parallel discrete simulator architecture consists of three components: (1) The parallel simulation, (2) a graphical user interface, and (3) communication libraries. The GUI is separate and perhaps on a different machine from the parallel simulation. The communication libraries (sockets, files or shared memory) are accessible to both the GUI and the parallel simulation. For simplicity and extensibility, character strings are used for the interface between components. The software architecture is shown in Figure 3.6.



Input String: <object>: <action><time stamp>:<gui_cb>:<uid>
Output String: <gui_cb> : <object> <value> : <time_stamp>

Figure 3.8: IPDES Architecture

The graphical user interface enables monitoring and steering of the simulation. The GUI is notified by the simulation which variables are available to monitor or steer during the initialization phase of the simulation. To query for a simulation variable x of logical process ID number 1 for instance, the GUI sends the string: "1 x:QUERY:" to the simulation. In general, the string is of the form: <OBJECT>:<ACTION>:<CALLBACK>. The <OBJECT> part of the string defines the object that is queried. An object is a logical process number ID and the name of the variable that is evaluated. Here the logical process ID is 1 and the variable name is x. The second part of the string defines the desired action. Sample actions are QUERY, MONITOR, MODIFY, SCRIPT, and CLONE, in our example, the action is a QUERY. The action can have parameters, e.g MODIFY would have a value associated with it. The action SCRIPT accompanied with an executable script enables an LP to modify its behavior. The third part of the string defines a call-back function that

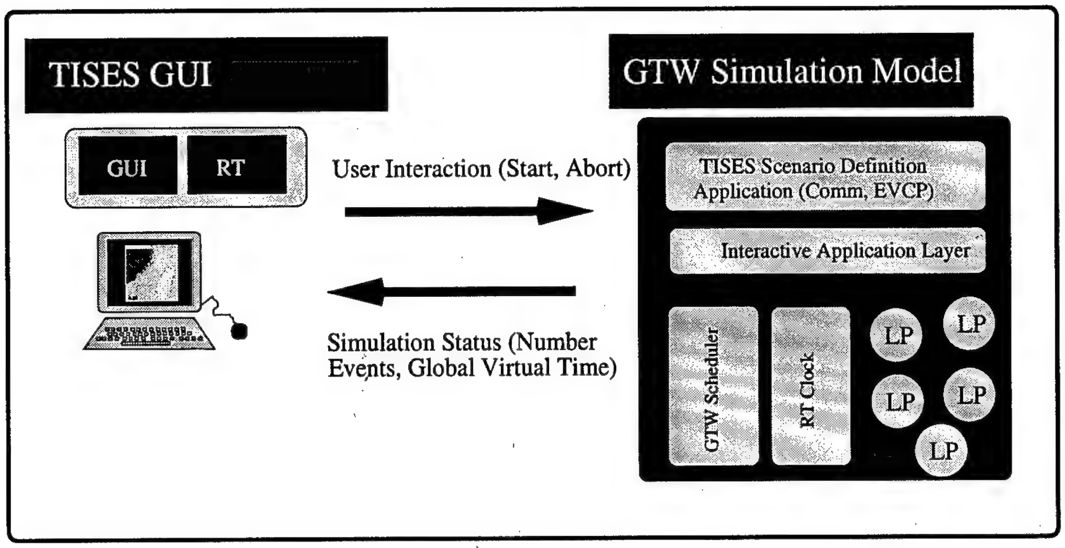

TISES GUI
GTW Simulation Model
GUI
RT
User Interaction (Start, Abort)
Simulation Status (Number Events, Global Virtual Time)
TISES Scenario Definition Application (Comm, EVCP)
Interactive Application Layer
GTW Scheduler
RT Clock
LP
LP
LP
LP
LP

Figure 3.9: TISES Interactive System Architecture

the GUI performs when the request returns. In this above example, no call-back action is requested.

The string interface between the components offers a flexible and extensible interface between the GUI and the parallel simulation. The GUI can show animated displays of performance metrics of the simulation. The GUI can easily be replaced by a textual interface program if needed, or a Java console. The design of the parallel interactive system is general, simple and offers transparent access to variables of the simulation.

3.7 System Architecture

This section describes the system architecture of the integrated TISES and GTW system for high-speed simulation. The architecture is composed of three modules: The TISES module, the GTW Simulation module, and the TISES Graphical User Interface module. Here, the TISES module is encompassed by the GTW Simulation module. The system is depicted in Figure 3.9. The system architecture differs slightly from the design above. The differences are highlighted below.

The TISES Graphical User Interface module communicates with the GTW Simulation module by sending a simulation start request or simulation abort messages via (rsh). These requests may be between different machines.

An interface to the TISES Graphical User Interface is an ASCII file called the Scenario Definition File (SDF). The SDF describes the missile positions during the run of the simulation. This file is pre-processed before the parallel simulation is instantiated. The TISES module (encompassed within the GTW Simulation Module, denoted the TISES Scenario Definition Application) in 3.9 is an emulation of the TISES system. The module emulates TISES by reading two ASCII files produced by an previous run of the TISES software. The first files describes the processing of TEWA events (See Section 3.3), the second files describes communication patterns between TOC's). GTW drives the simulation by scheduling TOCs in time-stamp order. Information such as the number of events processed and Global Virtual Time updates is send to the GUI periodically. The GUI can then show appropriate graphical display depicting the current state of the simulation system.

3.8 Graphical User Interface

Although the graphical user interface has custom-designed support for TISES events, a major portion of it can be used for general purpose Time Warp monitoring and steering tasks. The graphical

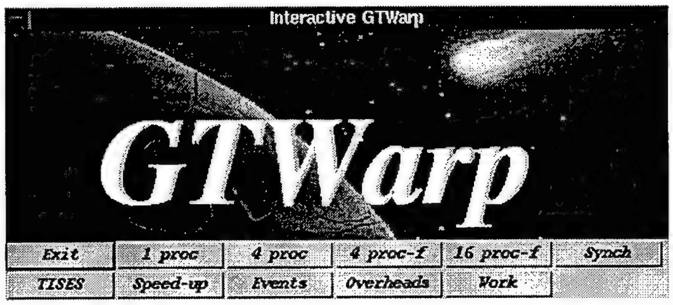


Figure 3.10: TISES Splash Screen

user interface is implemented in Java to ensure portability to many different platforms. When the TISES GUI is started, the initial control panel, sometimes called the splash window/screen, is launched. The splash screen is shown in Figure 3.10. The purpose for this window is to give the user control to launch a simulation with specific simulation parameters or to launch a status window to monitor performance metrics. The buttons in the top-row define different runs of the China-Taipei scenario (other runs can be defined by supplying a different ASCII file when the GUI is instantiated). The scenarios differ in the number of processors that are used. Currently, 1, 4 and 16 have been tested. A specific run is selected by pushing a button that defines the number of processors desired. A simulation is launched by using the Synch button. The bottom row launches different windows showing speedup, GTW overheads, amount of work completed, missile position updates, and number of events processed. Each window has two top buttons, a reset button and a close button. The Reset refreshes the shown image and and the Close button closes the window. Each of this windows are described in more detail below.

3.8.1 TISES China-Taipei Window

The China-Taipei scenario generates 57 threat objects, of which 36 are determined to be lethal. The display is shown in Figure 3.11 at different simulated times: 0, 350.88 and 1,500 simulated seconds. The purpose of this window is to give graphical positional information of missiles and threats as the simulation progress. The missiles are denoted by a small filled in circle, a history of its trajectory is platted as the missile progresses (see the middle image of Figure 3.11). In the rightmost image shown in Figure 3.11 the larger blue unfilled circle denotes that a threat missile and an anti-ballistic missile have collided. A red un-filled circle implies that the threat missile from China escaped the anti-ballistic missile and detonated. The China-Taipei window is launched by pushing the button marked TISES on the splash window.

3.8.2 Speed-up Window

The speedup-windows show the performance of the parallelized TISES with respect to sequential time (Sequential Time/Parallel Time). An example Speed-up window is shown in Figure 3.12. The

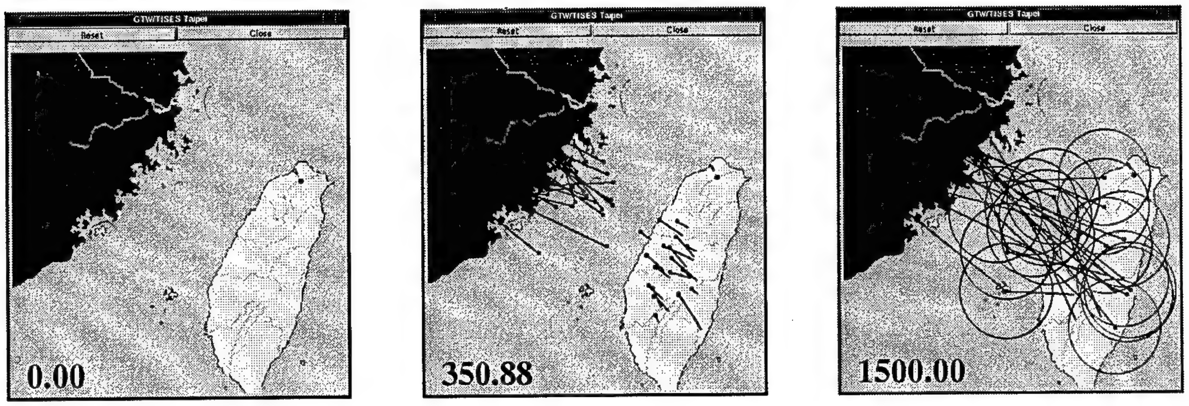


Figure 3.11: China-Taipei Window Animation Sequence

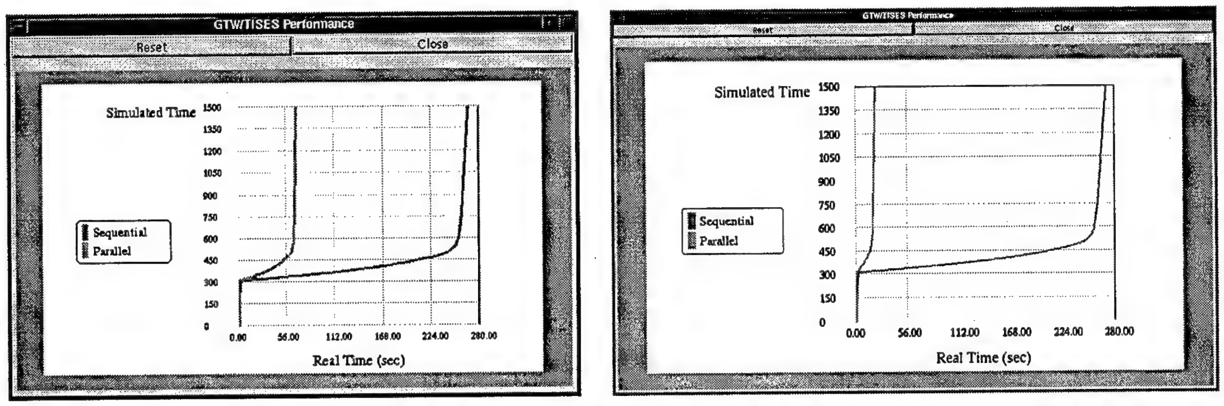


Figure 3.12: TISES Speedup Window, 4 processor on the left, and 16 processor on the right).

sequential run (the lower curve in each window) is not animated as the parallel curves, since the sequential run is pre-run. However, there is a parameter that can be set so that the sequential curve is animated as well, but then the curve simulates the sequential run (the sequential run is not actually run, but uses the result from a previous run). The parallel curve is green and the sequential curve is grey. Figure 3.12 shows a run of an emulation of TISES on an Origin 2000. The Speed-up window is launched by pushing the button marked Speed-up on the splash window.

3.8.3 Overhead Window

The Overhead window shows the performance of each processor as the simulation progresses. The performance of each processor is shown by two bar graphs, the leftmost illustrating number of processed event, or net events, and the rightmost showing overhead associated with the parallel run. An example of this window is shown in Figure 3.13. The number of processed events is sent from the simulation and is derived from subtracting overhead from a counter that maintains the total events processed on each processor. Total events includes committed events, rollback events, aborted events etc., hence it includes overhead that would not exist in a sequential run. The overhead thus constitutes the difference between the counter that maintains total events and total events that are roll-backed, aborted or cancelled. Net events are shown as a green bar, and overhead is shown as a red bar. The Overhead window is launched by pushing the button marked

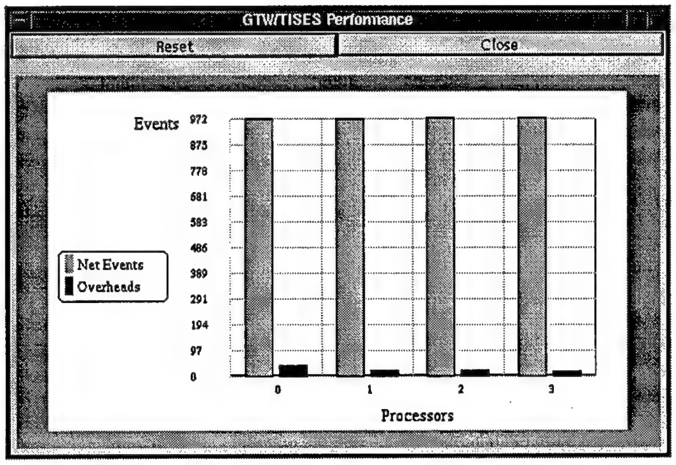


Figure 3.13: TISES Overheads.

Overheads on the splash window.

3.8.4 Events Window

The event window is similar to the overhead window in that it also shows net events and overhead. The difference is that fewer bar graphs are shown. An example of the event window is shown

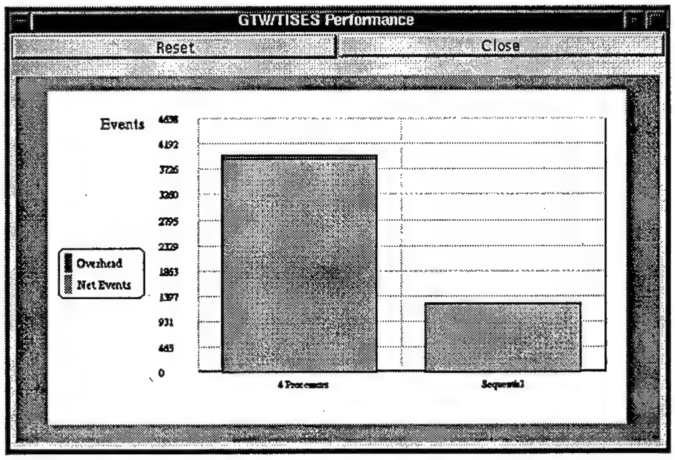


Figure 3.14: Events

in Figure 3.14. The purpose of this window is to get a feeling how the parallel run progresses compared to the sequential run. Both bars move as the parallel run progresses if the sequential run is simulated. If the sequential run is not simulated only one bar is shown, or the sequential bar is already at its maximum height. The bar on the left shows the net events processed by all processors and is marked in green (lower part of bar). The top part of the bar which is colored grey denotes the overhead, or number of events that are rolled back, cancelled or aborted. The bar on the right if shown, shows events processed by the sequential run. The right bar is all green since a sequential run will not have any overhead. The Events window is launched by pushing the button marked Events on the splash window.

3.8.5 Work Window

The work window shows amount of work completed as the simulation progresses. Work is equivalent to the amount of work that has been done with respect to work remaining. The display for the

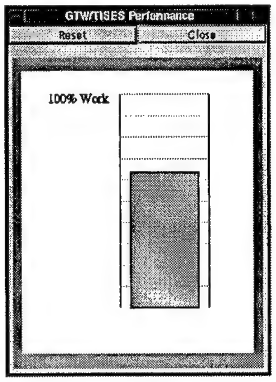


Figure 3.15: Work

work window is shown if Figure 3.15. The work bar is green and shown as a percentage. The Work window is launched by pushing the button marked Work on the splash window.

3.9 Conclusions

This chapter described the Interactive Parallel Discrete Event Simulator system and its integration with GTW and TISES. The GUI is implemented in Java to ensure portability to different platforms. Currently the interactive system provides a monitoring capability. Future work includes to provide steering and the cloning that was described in an earlier report.

We have developed and demonstrated an incremental approach to re-engineering an existing large-scale sequential battle management simulation. Because of the similarities between Ballistic Missile Defense (BMD) and National Missile Defense (NMD), particularly in BMC^3I arena, we believe our approach could be used to reduce the execution time of NMD applications as well.

Chapter 4

Optimistic Computations in Virtual Environments

The research goal of the I/O mechanism is three-fold: (1) to provide for high-latency computations in a real-time environment, (2) to enable the re-use of existing software by spawning specified computational modules and (3) to develop a framework to evaluate the effectiveness of the approach. Potential application domains include: interactive virtual environments for hardware testing (missile tracking sensors) that require high-fidelity image generation and domains like robotics where motor control commands in a multi-joint robot require a significant amount of computations, but where the presence or absence of obstacles may change which computations are necessary.

In order to provide an effective environment in these applications, computationally intensive modules (such as image generation, or robot kinematics) must be produced at real time rates. In the remainder of this report I will be referring to the virtual environment for illustrative purposes. A problem that crops up in virtual environment is that physically accurate images require at least seconds and sometimes hours to compute. It is clear such latencies are unacceptable. A solution to this is to use a "faster-than-real-time" simulator that can pre-schedule image sequences before they are needed. However, this approach may not be sufficient when users or devices *interact* with the environment leading to the requirement of different image sequences than those that were anticipated. A result is that there may be a significant lag between an external input and the appearance of the correct image.

This problem is addressed by leveraging two strategies used in optimistic parallel simulations: (1) speculative computations and (2) rollback to recover from an un-expected event. Specifically, a new mechanism called Optimistic I/O (OIO) is introduced to *speculatively* pre-compute image sequences ahead of real-time and to recover from external input by rolling back to a state corresponding to the point of interactions without disturbing portions of the simulation not affected by the afore-mention input. In addition OIO avoids I/O thrashing by limiting the extent of "optimism" in scheduling image requests according to a virtual time we call optimistic GVT

A prototype implemented on a shared memory multiprocessor employing off-the-shelf computationally intensive software, Synthetic Scene Generation Model (SSGM), demonstrates the application of this approach in a virtual environment.

4.1 Introduction

The high computational demands of phenomenologically correct image generation limits its use in real-time environments. These types of images often take minutes or sometimes hours to compute. This is a a problem when users or devices interacting with the environment require different images than those that were anticipated. A result is that there may be a significant lag between the interaction and the appearance of a correct image.

By initiating computations in advance, using a faster than real-time simulation system, one can overcome the latency in generating such image sequences. Abort operations may be activated if unexpected interactive inputs arrive to cancel time-consuming operations that are no longer needed.

Bringing virtual environments into interactive and dynamic federated simulations raises two challenges: One is to reduce the time required to generate individual images. The second is to hide the latency of performing these computations. The first issue is not addressed here. Even so, reduced image generation time alone is often insufficient to create realistic animation sequences because it may still not be feasible for real-time generation of frames. However, a faster-than-real-time simulation can execute ahead of time to determine what images will be needed later to hide the latency associated with image generation.

An optimistic parallel simulation system is well suited for this task. Parallel simulation techniques can enable many simulations to execute faster than real time. Further, optimistic simulators provide a roll back capability, so they can deactivate unwanted image generation that have been initiated but is no longer needed. Unexpected interaction can thus be dealt with when a Time Warp system is used in conjunction with the virtual environment. In this report we evaluate the effectiveness of this approach in a practical simulation application. Another motivation for this research is the *re-use* of existing software by independently spawning specified computational modules. The same capabilities may be used in other domains like robotics where motor control commands in a multi-joint robot require significant amount of computations, but where the presence or absence of obstacles may change which computations are necessary.

An Interactive Parallel Discrete Event Simulation System (IPDESS) is implemented to drive the generation of computationally intensive images for real time animations. A missile defense application is used to illustrate the approach. In this application the goal is to test missile defense strategies in realistic combat scenarios involving human-in-the-loop, hardware-in-the-loop, or computation only simulations. For example, an image sequence depicting a missile launch may be activated when a user clicks a mouse key. Subsequent images may include elements such as missile hard-body and missile plume. Another user may want to interact with the active animation in progress by pressing another button to activate an intercept missile. The images may include multiple missiles: the intercept missile and the target missile. Later in the simulation some images may depict either a hit or a miss of the intercepting missile with the target missile.

The chapter is organized as follows: The next section describes background and related work concerning interactive parallel simulations. Section 4.3 presents the optimistic I/O mechanism. This is followed by an example highlighting the benefit of our mechanism. Next, Section 4.5 discusses the implementation of the mechanism in a Time Warp executive. We then describe the off-the-shelf software used to generate images. The resulting software architecture and its implementation are discussed in Section 4.6.2 and 4.6.3. We conclude with a summary and a discussion of future directions.

4.2 Background

In an interactive animation environment I/O events direct the display of individual images; user interaction may initiate an unexpected image sequence. To illustrate the problem of integrating an off-the-shelf computer graphics engine into an action-oriented real-time environment, this report considers an animation package that require considerable computation time for individual frames.

To hide image generation times we exploit parallel discrete event simulation [Fujimoto 1990a] (See Chapter 2 for a review on parallel discrete event simulation). In a nutshell, distinct components in a parallel discrete event simulation are modeled by logical processes and the simulation progresses as LPs exchange time-stamped event messages that may cause changes in the system state at discrete points in time.

A synchronization mechanism is required to ensure that each LP processes events in time-stamp order. The two prevailing classes of synchronization protocols are called *conservative* and *optimistic* mechanisms. The conservative protocol enforces consistency by avoiding the possibility of an LP ever receiving an event from its past (as measured in simulated time). LPs *wait* to process events until reception of an out of order event is impossible [Chandy and Misra 1979; Bryant 1977]. The optimistic protocol, in contrast, uses a detect-and-recover scheme. When an event is received in the past of an LP it recovers by rolling back previously processed events with later time-stamps than the one that was just received. Since the optimistic approach has roll back capability, it can recover from unwanted computations such as image generation requests.

The prevalent optimistic approach utilizes the Time Warp mechanism originally presented in [Jefferson and Sowizral 1982]. LPs process incoming messages optimistically without blocking. If a logical process later receives a "straggler" message that has a lower time stamp than its current simulation time it rolls-back all processed events with a later time stamp than the straggler. The straggler and any rolled-back events are then re-processed in time stamp order. Roll-backs involve sending anti-messages that cancel or "annihilate" the original messages. To enable roll-backs Time Warp systems keep a copy of processed messages in an event queue. Messages are kept in the queue until they are ensured not to be subject of any future roll-back. A message is guaranteed not to be subject to a future roll-back if it is earlier than the smallest time stamp of any unprocessed or partially processed message in the system. This lower bound is referred to as the global virtual time or GVT of the system. GVT is used to commit I/O operations and to reclaim memory for events (called fossil-collection). Any event with a time-stamp lower than GVT is guaranteed to never be rolled back and can be committed, so its I/O operations can be performed and the memory for that event can be reclaimed.

Conventional Time Warp systems provide a mechanism for conservative I/O [Jefferson 1985]. We use a Time Warp executive called Georgia Tech Time Warp (GTW). In GTW two types of conservative I/O are available: nonblocking and blocking I/O events [Das et al. 1994]. A logical process can execute optimistically beyond a nonblocking I/O event because a nonblocking event does not effect the state of the simulator. Nonblocking events can be used for output events. Blocking I/O events however, prevent the optimistic execution of an LP. Because these events affect the state of the logical process, an LP blocks the execution of other messages with time stamps later than the I/O blocking event. The logical process cannot execute until the I/O event is either processed or canceled. Once GVT progresses beyond a blocking I/O event and the I/O event is processed the logical process becomes unblocked. Blocking I/O events can be used for input.

There is an important difference between committing an I/O event in Time Warp and displaying an image in an animation. Display events are related to real-time, but Time Warp I/O events are related to GVT. Since Time Warp is faster than real time, a new type of event, called an *optimistic* I/O event, is proposed to hide the latency of generating images. The idea is that optimistic I/O events can request an image before it is needed in real-time. The I/O events are referred to as optimistic because there are some cases (due to real time user input) where the scheduled events must be canceled. The simulation aborts time-consuming requests such as image generation that are no longer needed by activating anti-computations called roll-back handlers. The roll-back handlers are user defined. Thus, the actual display request of an image can be undone or changed. Unlike conservative I/O events which cannot be undone, optimistic I/O can be canceled or rescheduled. This will be discussed in detail in section 4.3.

In contrast to analytic simulations, virtual environments include either human interactions or hardware realizations as an integral part of the simulated system. Virtual worlds require real-time execution as opposed to analytic simulations that often proceed faster than real-time. Virtual environment simulations often exploit limitations of human perception. For example, events happening close together in time can often be processed in any order since a human cannot perceive the actual ordering. An example of the integration of an analytical protocol with virtual environments is the Aggregated Level Simulation Protocol (ALSP) [Wilson and Weatherly 1994]. The motivation behind the ALSP was to extend distributed virtual environments to the analysis of war-gaming. ALSP processes events in strict time-stamp order. This is accomplished using the Chandy/Misra/Bryant conservative protocol mentioned earlier.

Interactive parallel discrete event simulation work by [Franks et al. 1997; Ghosh et al. 1993] differers from ours in that they do not use compensatory operations to prevent long-running operations. Further we deliberately avoid the phenomena of I/O thrashing described in the next section. In databases, compensatory transactions are used as an alternative to strictly un-do the effect of long-running operations while avoiding cascading rollback [Garcia-Molina and Salem 1987; Korth et al. 1990]. Our approach is inspired by compensatory operations in databases to enable the use of off-the-shelf software in virtual environments. We use optimistic simulation techniques to hide the latency associated with time consuming operations. The next section gives details of our approach.

4.3 Optimistic I/O: Approach

Figure 4.1 shows the mechanics of a Time Warp simulation. The right image shows the communication topology of logical processes (LPs) participating in the simulation. The left image shows the progress in simulated time of the LPs. The vertical axis is simulated time, the horizontal axis represents each logical process (except the lower leftmost dark grey rectangle marked "RT" which show the progress of wall clock time.) This graph relates the simulated time of the Time Warp system to wall clock time. GVT is defined as the lowest time stamp of any unprocessed message or partially processed message in the system. Since a Time Warp simulator progresses faster than real time, wall clock time is behind GVT. If a user is allowed to interact with the simulation, the event that represents the interaction should have a time stamp equal to wall-clock time. This would push GVT to an earlier time. In this situation GVT can not be used for fossil collection since it is possible for a rollback to occur that is earlier than GVT.

A simple solution to realize optimistic computing in a real-time environment is to include a new LP that paces interactive operations according to the real-time clock. Here, the simulator (excluding

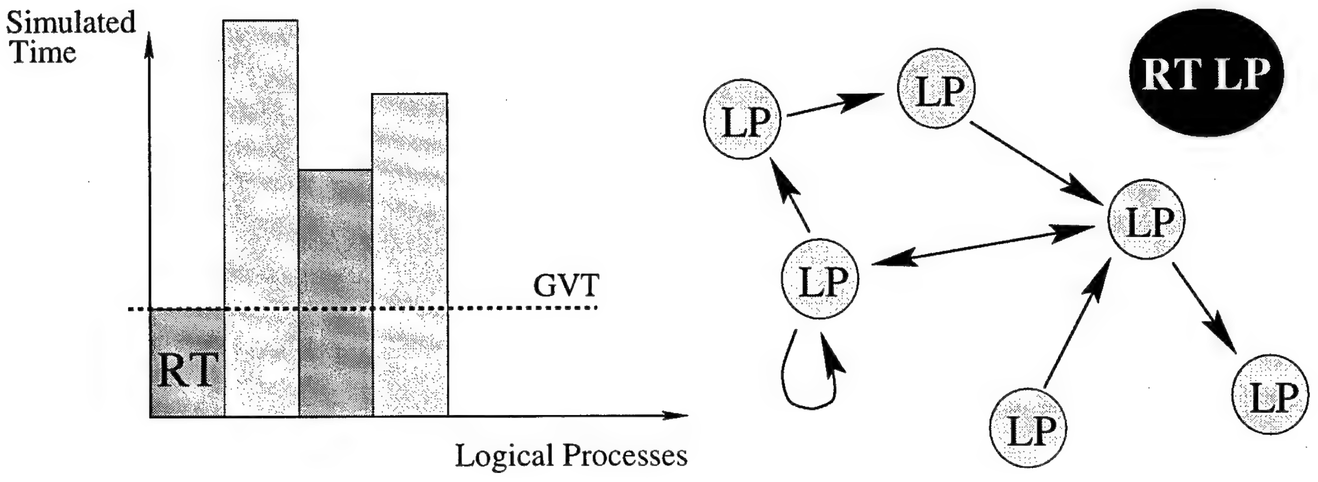


Figure 4.1: A Simple Solution for Real-Time Time Warp, each bar represents the progress in simulated time of a logical process in the simulation.

the new LP) is faster than real-time. As a result the new LP encapsulating interactive input/output includes the real-time clock and suspends GVT accordingly. The effect is that interactive events are released at GVT while other operations can proceed optimistically. This however, does not solve the problem with computations that are computationally intensive. So a second solution is needed.

To provide for computationally intensive operations the better approach provides a compensatory operation. Here, computationally intensive operations are allowed to be scheduled before its time stamp reaches GVT. However, if it is found that the operations are no longer needed a compensatory operation that aborts the request is activated. This is accomplished by piggy-backing a user-defined abort operation on the event that is instantiated upon rolling back the operation. There is a problem with this solution as well. The problem is that there may be a high occurrence of requests that are later aborted resulting in I/O thrashing. The solution can be improved to limit the optimism of these types of transactions.

To avoid I/O thrashing two GVTs are defined. One is called conservative GVT and the other is called optimistic GVT. Figure 4.2 illustrates the different GVTs. *Conservative* GVT is defined to be the minimum of wall-clock time and the current GVT. An event is fossil collected when its time stamp is lower than the conservative GVT since these event are guaranteed to never rollback. *Optimistic* GVT is the minimum of any unprocessed message or partially processed message. Optimistic GVT is the same as the conventional GVT in a Time Warp system when the LP encapsulating the wall-clock is excluded. Mirroring the GVTs there are conservative and optimistic I/O events. Conservative I/O events are executed as conservative GVT progresses beyond their time stamps. Similarly, optimistic I/O events are executed as the optimistic GVT sweeps their time-stamps.

With respect to a virtual environment, an image request message is an optimistic I/O event, it is buffered until its time stamp is beyond the optimistic GVT. When optimistic GVT catches up to these buffered messages they are processed and the image request is sent to the virtual environment coordinator. When a real-time user-input request is processed, a message is sent to the Time Warp simulator. The simulator responds by rolling back events that have a time stamp later than conservative GVT (includes wall-clock time.) If an optimistic I/O operation is undone, it aborts its corresponding image generator invocations by activating the anti-computation procedure associated with the optimistic event.

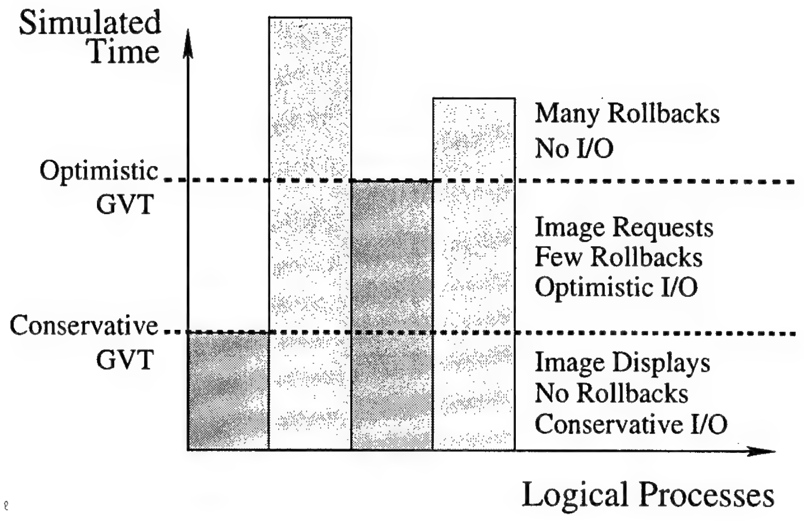


Figure 4.2: Optimistic and Conservative GVT

4.4 Example Task

To illustrate how an optimistic simulator can improve the performance of a physical simulation consider the task of intercepting a target missile. The intercept of a target missile has an unpredictable element, namely a hit or a miss. The unpredictable element can be introduced to the interactive animation by providing a push button to launch a target missile. An illustration of the scenario is shown in Figure 4.3. The idea is that a user pushes a button to activate the launch of a target missile. Next, another user activates an intercept missile aimed for the target missile. The result is either a hit of a miss depending on when the users pushed their buttons. When the user pushes

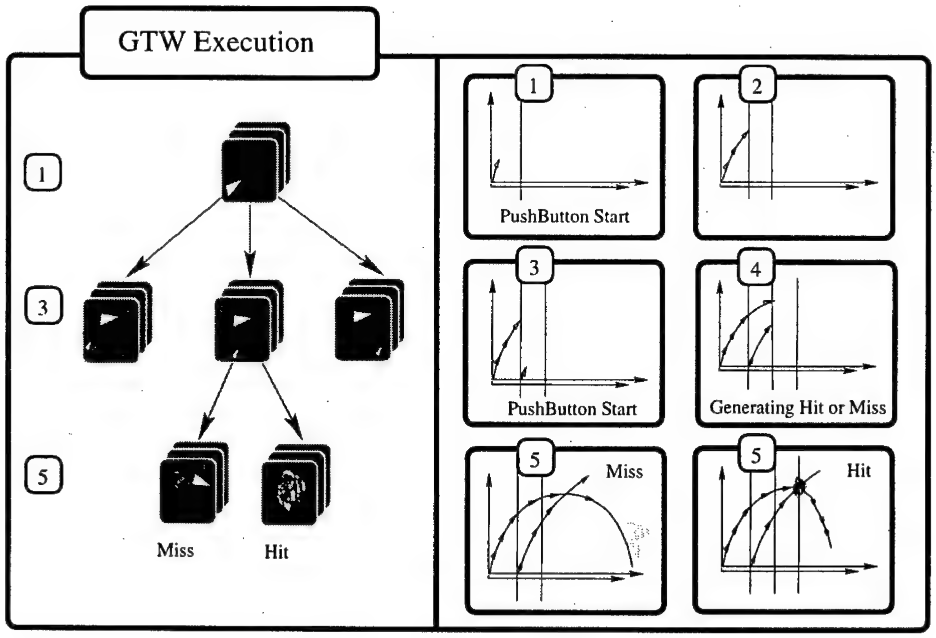


Figure 4.3: Example Scenario

the button to launch the intercept missile, Time Warp will first cancel images in progress that do not contain the intercept missile. Also Time Warp will start requesting images that contain both missiles. As the simulation draws closer to the concluding image (the hit or miss,) Time Warp requests the generation of images which display either a hit or an miss. If a hit occurs, the most likely one, a miss, may already be in progress and thus is canceled. When the actual event occurs and is displayed however, the hit or the miss image will already have been generated and can thus be immediately displayed.

4.5 Optimistic I/O Implementation

The optimistic I/O mechanism is implemented using Georgia Tech Time Warp (GTW). While discussing the implementation, it is helpful to introduce a few details of GTW. GTW provides procedures for message passing, for example non-I/O messages utilizes the `TWSend` procedure. In implementing optimistic I/O messages GTW provides a `TW_OIOSend` procedure. `TW_OIOSend` differs from `TWSend` in that a different event handler can be specified for each event. This is similar to the two existing conservative send primitives, but `TW_OIOSend` has two event handlers associated with it, one which is activated when the event is processed and one that is activated if the event is rolled back or canceled. With respect to the example application the `TW_OIOSend` procedure can be defined to "request" the generation of a image with specified parameters. For example the parameters may specify the take-off of a target missile at a certain geographical location.

As previously mentioned I/O messages are scheduled for processing when their time stamps are earlier than GVT. In contrast to conservative GVT optimistic GVT is not a monotonically increasing function, because an external interaction may set back the optimistic GVT to an earlier time stamp.

In order to delay the processing of optimistic I/O messages until their timestamps are earlier than the optimistic GVT, each processor maintains a priority queue holding unprocessed conservative nonblocking I/O events and unprocessed optimistic I/O events. I/O events are added to this queue as they become the next pending event. A procedure which is called immediately after a new optimistic GVT has been computed, ensures queued optimistic I/O events with time stamps less than or equal to the optimistic GVT are processed. The processed message is then put in the respective LP's processed event queue in case the message must be rolled back. If a processed optimistic I/O event is rolled back a rollback function for that event is activated. GTW refers to the parameters of the original processed event to the rollback function. Hence, resources can be conserved in the interactive environments if the users specifies that image generations are canceled when their corresponding events are rolled back. The parameter to the rollback function in this case would be the corresponding instance that requested the generating the image. The algorithm is listed below:

1. LP sends optimistic I/O message via `TW_OIOSend` primitive.
2. The sending LP of an optimistic I/O message enqueues the message directly into message queue corresponding to the destination LP (GTW provides one message queue per processor.) If the destination and sending LP are mapped to the same processor the message is enqueued directly into a pending queue.
3. If a processor dequeues an optimistic I/O event from its pending queue the event is put in the processor's optimistic I/O holding queue.

4. When optimistic GVT is computed, the events in the I/O holding queue are processed if their timestamps are earlier than the optimistic GVT.
5. If a processed optimistic event is later canceled or rolled back, due to an external event, a specified rollback function is activated.

GTW launches a separate thread called EX to monitor external input events or to output an external event. When a user pushes a button for example, the request goes through the EX thread. In turn the EX thread routes the external event to the appropriate LP.

4.6 Evaluation

OIO is evaluated by demonstrating the (1) feasibility and the (2) effectiveness of the approach. The feasibility of optimistic I/O is demonstrated by developing a prototype using the Synthetic Scene Generation Model (SSGM) software package developed by the Naval Research Laboratory (NRL). The prototype and SSGM is described in the next section. Effectiveness is evaluated by examining the effect of the quality of the animation sequence as the probability of user interaction increases and is described in section 4.6.4.

4.6.1 Synthetic Scene Generation Model

In order to demonstrate the capabilities of optimistic I/O the Synthetic Scene Generation Model (SSGM) software package developed by the Naval Research Laboratory (NRL) is used to generate images for a virtual environment. SSGM generates high quality image sequences for Ballistic Missile Defense Organization (BMDO) scenarios. SSGM is important to researchers because it produces images that are physically correct with great accuracy, such accuracy is needed because the same set-up is used to test actual sensors (hardware-in-the-loop). The computed trajectories, plumes and clouds have complex mathematical and time consuming algorithms associated with them. Commercial graphic packages can not generate images near the quality of SSGM images. SSGM produces physically correct optical images for development of Strategic Defense Initiative Hardware (SDI). SSGM is primarily used to predict the performance of optical sensor systems that are designed to track the detection, acquisition and discrimination of targets.

An SSGM image contains three elements, the background, the foreground and the observer element. The elements are specified via parameters in an ASCII file called the Scenario Definition File (SDF). Several phenomenologies exist for each element. For example, a user can select among several missile and plume phenomenologies for the foreground element. Background components may be terrain from various geographical locations or/and specific cloud formations. In contrast to the background and foreground elements the observer element has one single phenomenology, called the Observer. Location and framing mode are parameters among others that must be specified for the Observer. The interface to the SSGM server is thus the Scenario Definition File which parameterizes the use of the phenomenologies.

Even though SSGM provides high-quality images, it cannot be used effectively in a real-time environment because:

- image generation takes to long (high latency).

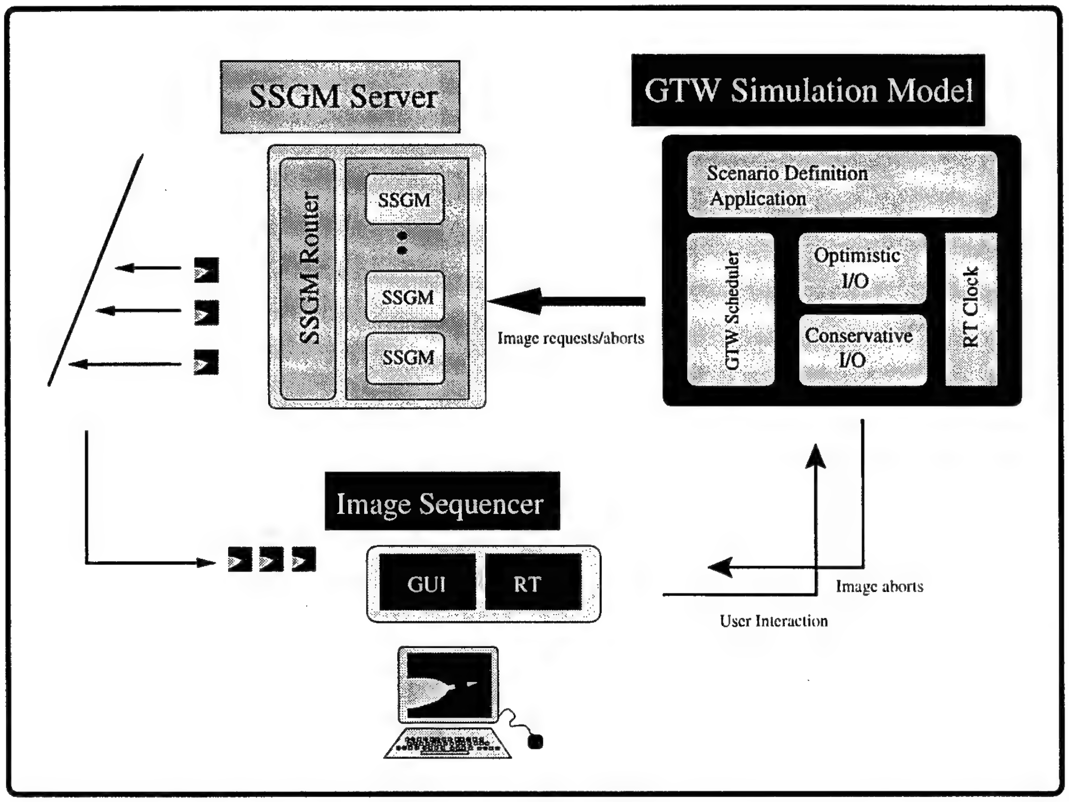


Figure 4.4: The Interactive Parallel Discrete Event Simulation System

- image sequences must be specified in advance and cannot be changed once the simulation has started. This inhibits scenarios such as missile intercept to be created dynamically by a user or hardware.

The optimistic I/O mechanism primarily addresses the latter issue.

4.6.2 System Architecture

This section describes the system architecture of the integrated SSGM and GTW system for high-speed simulation. The architecture is composed of three modules: The SSGM Server module, the GTW Simulation module, and the Image Sequencer Server module. The system is depicted in Figure 4.4.

GTW communicates with the SSGM module by sending image request or image abort messages. Image abort messages cancel images which may be in progress but no longer needed. Image request messages are sent as a result of an optimistic I/O event in Time Warp. The events are "optimistic" because they request images likely for future display. There are situations, however, when the predictions are wrong. In these situations the simulation needs to undo the incorrectly predicted event otherwise computing resources may be wasted. Time Warp preempts the image generation by activating a recuperate event which aborts the respective unwanted SSGM computation. Note that the ability of an optimistic I/O event to recover from unnecessary output results further differentiates optimistic I/O events from conservative I/O events.

The interface to SSGM is an ASCII file called the Scenario Definition File (SDF). The SDF

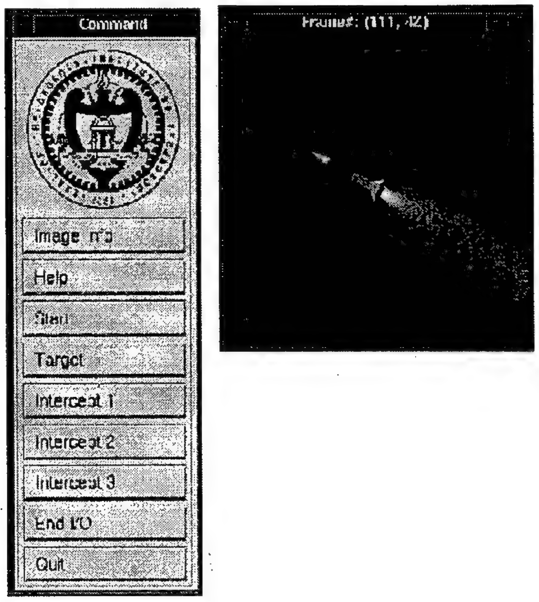


Figure 4.5: The Image Sequencer Interface

describes the image phenomenologies by parameter. An optimistic message thus includes a scenario definition file generated by the GTW executable. The delay in producing an image within SSGM can be reduced by parallelizing the computation of each phenomenology. This effort has been undertaken by NRL. In our research at Georgia Tech we consider SSGM as a black box, accessible via sending request or abort messages for image generation.

The image sequence module is responsible for displaying the correct image at the correct time. This is done by encoding the frame number into the name of the resulting file.

GTW drives the simulation by sending requests for image generation to SSGM. If a user interacts with the simulation, an external event is sent to GTW. GTW responds by aborting the corresponding optimistic I/O events and by requesting new image generation reflecting the user's interaction.

Image Sequencer Module

The image sequencer module is responsible for showing the correct image at the correct time. It also provides the user with push buttons that allow interaction with the simulation in progress. The current prototype has six possible interactions for the example task. Namely: START simulation, LAUNCH target, LAUNCH interceptor 1 through 3 (one launch for each interceptor), and STOP interaction. Figure 4.5 shows the user interface. When a user pushes any of the interaction buttons a message is send to GTW through a socket. The message contains the object (intercept, target or simulation), the action desired (start or stop,) and the current time. Time Warp interprets the message and then starts to generate future scenario definition files which accounts for the interaction.

SSGM Simulation Module

Even though we consider the SSGM as a black box, we have extended the SSGM module so that it behaves more like a server. The current model requires that only one scenario at a time can be processed or queued. In other words, SSGM does not queue requests. Further, if an image abort is desired a whole SSGM process has to be aborted even if only a few images are not needed. In order to circumvent the queuing problem we added a sub-module that we call the SSGM scheduler. The SSGM scheduler, takes the requests from GTW then forwards the requests to an available machine. If no machine is available it queues the request and waits until a machine can fulfill the request. If an abort is requested however it looks in its queue and removes it if it is in the request queue otherwise it aborts the appropriate SSGM process in progress.

4.6.3 IPDESS Implementation

The Interactive Parallel Discrete Event Simulation System (IPDESS) was implemented using the Georgia Tech Time Warp version 2.5 (GTW_v2_5), the Synthetic Simulation Model version 6.0 (SSGM_v6_0), ImageMagick (IM), and a GTW scene parameter application. This section will describe the implementation of I/O operations in the GTW executive, the scene parameter application operating on top of GTW, and the graphical interface that provides for user interaction. GTW is the central manager of the overall system. It drives and controls the animation in progress by communicating with each module of the IPDESS system. Conversely, the modules communicate with GTW to request actions in the system.

The goal of reducing the image latency involves solving several implementation issues in the GTW kernel:

- Provision for optimistic and conservative I/O
- Provision of an optimistic rollback routine for each optimistic I/O event

The goal of the IPDESS is to hide image generation latency of SSGM. In it's current implementation however SSGM may take several hours to generate an image (efforts to reduce this are underway at NRL.) In anticipation of a faster SSGM, the IPDESS can use precomputed images with a delay to simulate processing, or if necessary, trigger actual SSGM computations.

The architecture of IPDESS was introduced in section 4.6.2. The connections between the Image Sequencer, the SSGM scheduler and GTW are sockets. For example, when GTW is initiated it launches the SSGM scheduler, and sets up the connection between them. It then waits for a connection to of an animation client (or the image sequencer.)

SDF Application Implementation

The SDF application is written using GTW Kernel primitives. The application uses for example `TWInitAppl`, `TWSend` and `TW_OIOSend` primitives. The application also defines the following event handlers: a target handler, three intercept handlers and a collector handler. Each of these handlers are associated with a Time Warp LP. The collector, target and interceptor each reschedule themselves for each frame period. If a target or an intercept receives a message from EX it changes it state to reflect the user request and then forwards this information to the collector. The collector sums the state of each object in the scenario, and sends optimistic requests to schedule SSGM instances. The collector also sends non blocking conservative I/O messages to ensure that the image sequencer and GTW are synchronized. The optimistic event handler schedules an SSGM event if there has been a change in state by sending a message to the SSGM scheduler, the event requests SSGM should, from now on, generate images reflecting the current specified state. If there is no change in state, the event handler returns. The reason that we trigger the optimistic event even if no request is send to the SSGM scheduler is to give a handle to individual frames in the event of a roll back. The roll back handler can thus can eliminate individual frames and tell the SSGM scheduler which images need to be aborted.

Image Sequencer Implementation

The interface for user interaction with SSGM simulation is provided in part by a public domain animation software package (ImageMagick). The package is enhanced so that it displays the correct image according to a real time clock. Buttons are provided to initiate target and intercept missile launch: as a button is pushed, a message is sent to alert the GTW module. The button pushes trigger or roll back optimistic I/O events in GTW. In order to address the high latency of SSGM, the image sequencer currently uses a library of precomputed images. The images are accessed by the state and the frame number of the scene. By state we mean all objects that could possible be visible. For example the target has not launched and there are currently no interceptor and the user has pushed the START simulation button the state is: SIM START, NO TARGET, NO INTERCEPT1, NO INTERCEPT2, NO INTERCEPT3. The image sequencer thus keeps track of its state and display the image that corresponds to the current state. As SSGM latency becomes reasonable and the images are available as they are generated, the animation tool need only undergo minor changes to show newly generated image instead images from the image library.

4.6.4 Effectiveness Evaluation

The effectiveness of the approach is evaluated by quantifying percent on-time coverage while varying the probability of user interactions (ρ). Coverage measures on-time delivery of images as a function of:

- the probability of user interactions,
- delivery period (1/frame rate),
- decision opportunities (the polling period for user input) and
- computational delivery.

It is assumed that these parameters are periodic. Coverage also depends on the "acceptable delay" which is the window between a user input and its temporal effect on the animation sequence. For example, consider an SSGM scenario where a sensor may track a threat missile, and an unanticipated intercept missile is launched. The effect on the animation sequence representing the field of view of the sensor is not effected until the intercept missile is in its field of view, which may be several seconds or minutes after the external input.

For completeness, % *coverage* in an animation sequence is defined below:

$$\%\ coverage = 100 * \left(1.0 - \frac{\text{total number of missed display events}}{number\ of\ display\ events}\right)$$

Clearly, total number of missed display events is the sum of missed events at each interaction. Missed events due to an interaction ($missed_{per\_input}$) depends on the following parameters:

$$T_{delivery} = \text{the delivery period}$$
$$T_{decisions} = \text{decision opportunity period}$$

$$\begin{aligned}
T_{comp} &= \text{computational delivery period} \\
delay &= \text{acceptable delay} \\
\text{recovery cost} &= \text{time to recover from an un-anticipated input}
\end{aligned}$$

where the relationship is:

$$missed_{per\_input} = \frac{(T_{comp} + \text{recovery cost}) \text{ - } delay}{T_{delivery}}$$

The total number of missed display events ($missed_{total}$) can be predicted from the probability that the user interacts at a decision point, and the total number of decision opportunities ($decision_{total}$) encountered. Recall, that ρ is the probability the user interact at a decision point, therefore:

$$missed_{total} = decision_{total} * \rho * missed_{per\_input}$$

The number of display events and decision opportunities can of-course easily be computed from the elapsed time of the animation sequence, and their respective periods.

To illustrate the benefit of using speculative computing (OIO) in an interactive environment the % coverage of a non-speculative simulator is compared against the OIO scheme. Coverage is evaluated by varying both the probability of a user interacting at a decision point and decision period with respect to computation time ($T_{decisions}/T_{comp}$), increasing the number of decision points per computation implies the fraction decrease. The non-speculative simulator can only pre-schedule *anticipated* computations, but it does not induce an additional cost, due to rollbacks, when scheduling images after an interactions. The recovery cost for the OIO scheme is evaluated at 10 % of the computation time, which is an over-estimation since computation time, e.g. SSGM image generation time, is high compared to the recovery cost of our parallel simulator GTW.

The % coverage of a speculative and a non-speculative simulator is shown in Figure 4.6. Speculative computing is advantageous when there is a low probability of interaction at a decision point and the computational period is large with respect to the decision period, i.e. the density of decision opportunities is high when the ratio $T_{decisions}/T_{comp}$ is low. In the plot this region is in yellow. This is precisely the characteristics of the SSGM: the probability that the user interacts is low, because there are few interceptors to launch in response to a threat, the computation to compute an image is high, and while it is desirable to poll for input frequently. In-fact for a non-speculative simulator the coverage is 0% when the polling period is equal or less to the computational period. Even when relaxing the polling period the speculative simulator outperforms the non-speculative by a significant factor when the probability of interaction is .5 and lower.

4.7 Conclusions and Future Work

This chapter introduced an optimistic I/O mechanism able to hide latency in application software by scheduling processing long before it's output is needed. A rollback computation is coupled with optimistic request to recovery of resources if an unexpected real-time input forces the cancellation of the scheduled operation. The mechanism enables the re-use of existing software by independently spawning specified computational modules.

OIO is evaluated by demonstrating the (1) feasibility and the (2) effectiveness of the approach. Feasibility is demonstrated by integrating a computationally intensive image generation application

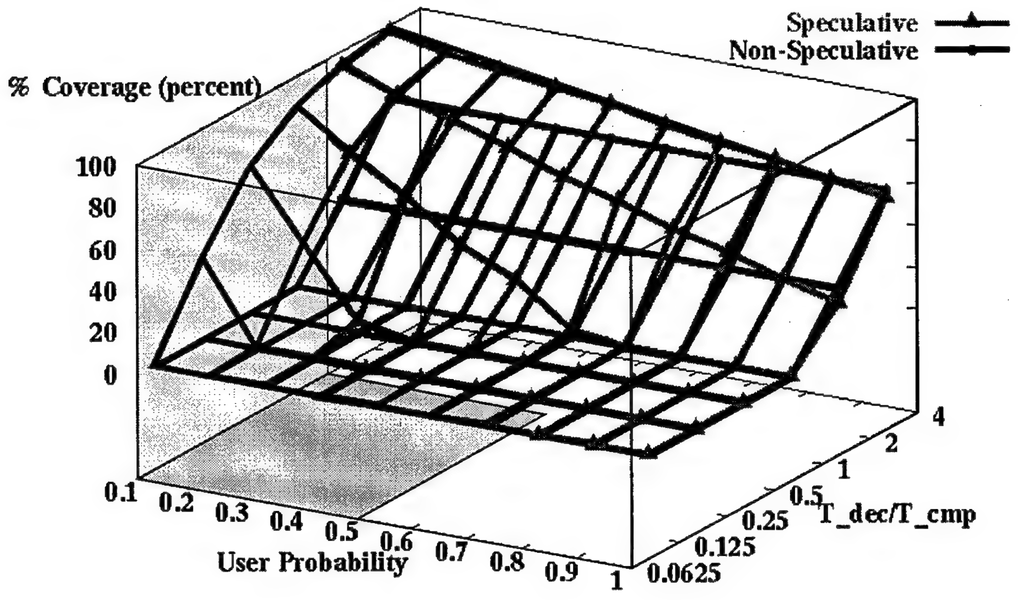


Figure 4.6: %Coverage

called SSGM into an interactive environment. A software architecture called IPDESS is proposed and implemented and contains three modules: the SSGM module, the GTW module and the image sequencer module. Effectiveness is evaluated by examining the effect of the quality of the animation sequence as the probability of user interaction increases. The observation from evaluating *coverage* of OIO, a speculative approach against a non-speculative approach suggest that speculative computing is advantageous in a virtual environment where the polling period for user input is high, while the probability of interactions is low (.5 and lower). This suggests that for an application such as SSGM the approach is viable.

Improvements to IPDESS include the implementation of a lazy cancellation scheme that reduces computation when rolling back optimistic I/O events.

Chapter 5

Cloning Parallel Simulations: A Programmer's Manual and Specifications

This chapter describes the Clone-Sim application programmer interface (API), and how to use it for simulation cloning. The next chapter gives details of the underlying mechanisms and performance of Clone-Sim. Clone-Sim enables on-demand "cloning" of parallel and distributed discrete event simulations. The package can be used in interactive as well as non-interactive environments. Both optimistic [Jefferson and Sowizral 1982] and conservative simulators [Bryant 1977; Chandy and Misra 1979] can be supported. Currently, Clone-Sim has been implemented with Georgia Tech's Time Warp simulation executive [Das et al. 1994] called GTW, an optimistic simulator.

Cloning is a mechanism that enables the concurrent evaluation of multiple simulated futures. The approach has been developed for parallel discrete event simulators, where the simulation consists of a collection of logical processes (LPs) potentially executing on different processors [Fujimoto 1990a]. These types of simulators traditionally have two types of primitives: (1) *send and schedule* which schedules an event on some logical process (ScheduleEvent) and (2) *receive* which processes a scheduled event (ProcessEvent).

A running parallel discrete event simulation is dynamically cloned at *decision points* to explore different execution paths concurrently. A decision point is where the states of different versions of a simulation begin to diverge. A decision point is defined or inserted *on* a specified logical process or processes and in this manner enables the exploration of different scenarios. The *user views* the whole simulation domain replicated into many different planes where each plane is an independent version of the simulation executing in parallel with the other cloned versions (See Figure 5.1).

The Clone-Sim implementation avoids the cost of brute-force methods that replicate an entire simulation. Cloning in Clone-Sim uses an incremental update scheme. In this paradigm each plane or *version* of the simulation contains a collection of virtual logical processes. A collection of virtual logical processes is created each time a simulation is cloned. The difference between the cloned simulations is in the mapping of virtual logical processes to physical logical processes. Here, a physical logical process refers to the run-time environment of virtual logical processes. Each virtual logical process (V) is assigned or *mapped* to a physical logical process (P). The idea is that several virtual logical processes can share the same physical logical process thereby avoiding replication of common computations. But two physical processes cannot be mapped to the same virtual process. An analog is virtual memory where the same main memory address can shared by several virtual addresses, but two addresses in main memory cannot be mapped to the same virtual address. The mapping between virtual and physical processes is updated as the clones diverge. Resources are

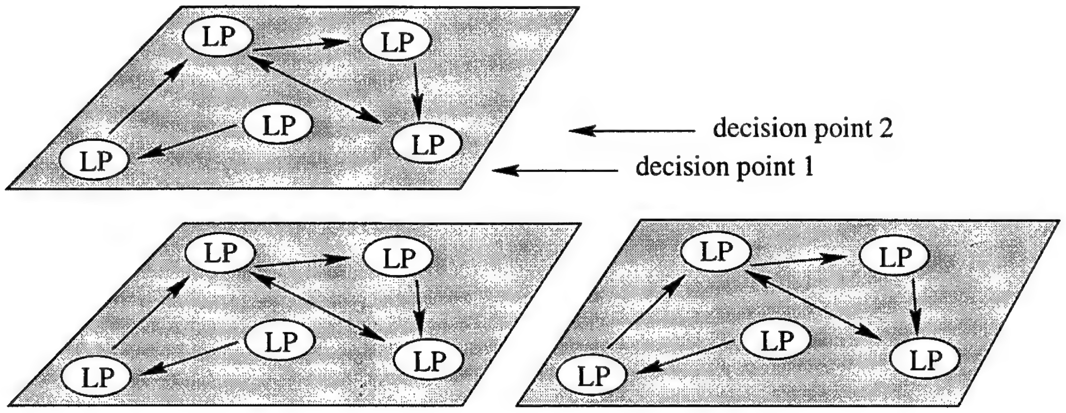


Figure 5.1: A Parallel Discrete Event Simulation Represented by Logical Processes (LPs) Replicated Twice; the Upper Left Plane Shows the Original Simulation

re-used as long as possible and only the smaller portions which cannot be shared are replicated.

To illustrate, the bottom image in Figure 5.2 shows the mapping between virtual processes and physical logical processes after the simulation is cloned on physical process A. Here the mapping of the original version of virtual processes stays the same and the computation between clones is shared. Virtual processes B and C, version one and version two share the same corresponding physical process; while virtual process A version one and version two maps to different physical processes.

As the simulation progresses the mapping of virtual processes to physical processes changes, as new physical processes are created. Message sends and receives are carried out in the physical layer. In this manner a physical send corresponds to a *set* of sends in the virtual process layer. For more details on the incremental update scheme and its performance see [Hybinette and Fujimoto 1998].

Clone-Sim achieves efficient cloning by intercepting the communication primitives of a simulator executive. By monitoring the send and receive primitives Clone-Sim can avoid unnecessary cloning of logical processes (LPs). Likewise, it can determine which logical processes need copies of messages. The key idea is that the receiving LP can determine whether to clone or forward by inspecting bits that are piggy-backed by the cloning mechanism on messages.

This manual is organized as follows. The following section describes the design goals of Clone-Sim. The underlying assumption are discussed in Section 5.2. Section 5.3 discusses the software architecture and the interactions between the software modules. The application programmer interface of Clone-Sim is described in Sections 5.4 and 5.5. The compilation of Clone-Sim and header file requirements are discussed in Section 5.6. Section 5.7 describes illustrates the use of the cloning primitives by describing an simulation that utilizes cloning written for the Georgia Tech Time Warp executive. Section 5.8 contains a reference manual.

5.1 Design Goals

Goals for the Clone-Sim implementation are:

- efficiency,
- transparency and

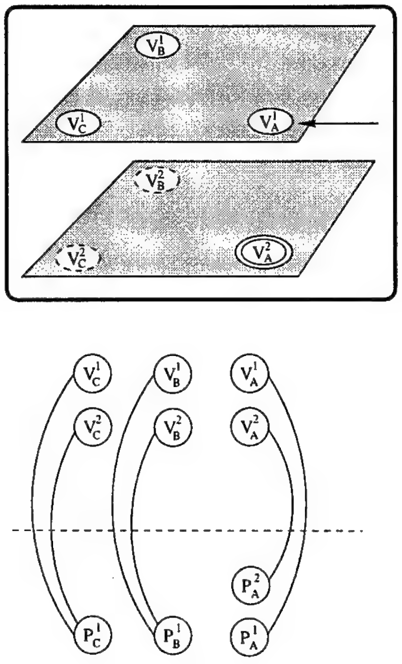


Figure 5.2: A Snapshot of a Simulation That Has Been Cloned; the Top Image Shows the Two Virtual Versions of the Simulation, the Bottom Image Shows the Mapping of the Virtual Processes to Physical Processes A, B and C

- simulator independence

Efficiency is in terms of number of alternatives evaluated in a time-constrained period and memory resource usage. Efficiency is achieved by enabling multiple scenario analysis and allowing different versions to share computations between themselves. Transparency is with respect to the simulation application and is accomplished by monitoring pre-existing primitives (*send* and *receive*). Simulator independence refers to the choice of optimistic or conservative synchronization. Here, Clone-Sim provides simulator independence with respect to this framework.

5.2 Assumptions

Clone-Sim is based on the assumption that the simulator executive provides *send and schedule* and *receive* primitives to the simulation application. The relationship between the simulation executive, the user application and the ScheduleEvent (send and schedule) and ProcessEvent (receive) primitives are illustrated in Figure 5.3. Here, the simulation application defines the events and the simulation executive manages the synchronization. For example if the simulation uses GTW, the application defines ProcessEvent, tells GTW when to schedule the event, then GTW actually makes sure that ProcessEvent is called when appropriate.

Clone-Sim also assumes that the simulation executive provides for dynamic LP creation including allocation, initialization and copying LPs. It is assumed that one can schedule events conservatively, i.e. the event can be scheduled with the assumption that it will never roll back, (this is trivial if the simulation executive is conservative). To summarize, Clone-Sim assumes:

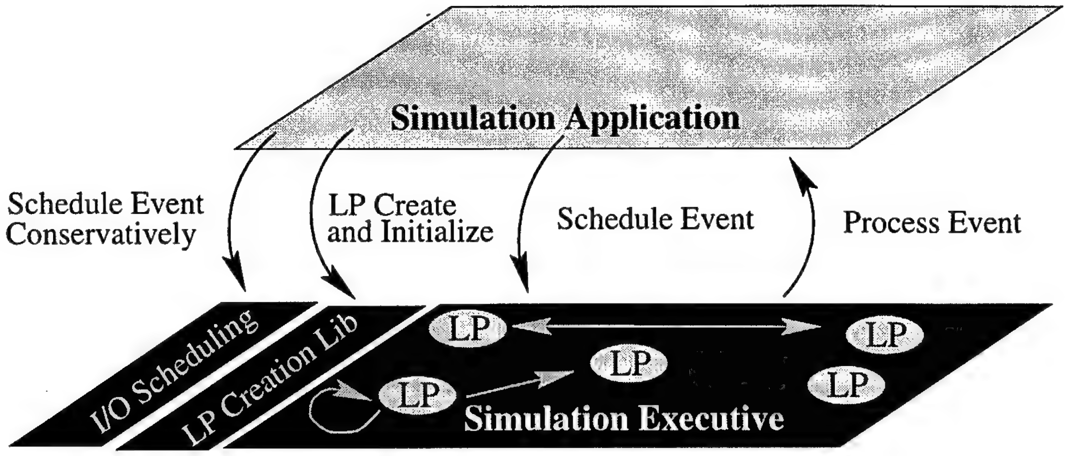
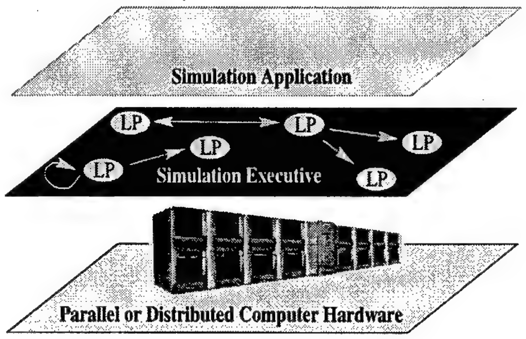


Figure 5.3: Clone-Sim Assumptions

- a *send and schedule* primitive (ScheduleEvent)
  - including capability to schedule an event conservatively,
- a *process event* primitive (ProcessEvent) and
- a capability to create/copy logical processes

5.3 The Clone-Sim Application Programmer Interface

A simulation consists of a simulation application (provided by the user) and a simulation executive that implements the synchronization protocol. The simulation executive provides primitives that allow simulation programmers to define their own applications. This is a layered system, with the operating system at the bottom, the simulator executive in the middle and the simulation application at the top (See Figure 5.4). Clone-Sim consists of two modules: the **Interactive-Sim** module which



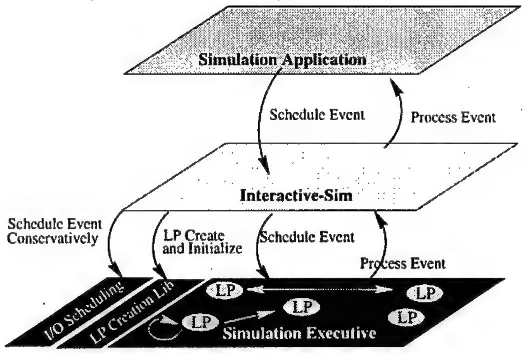


Figure 5.4: Views of Simulations: Traditional Parallel Discrete Event Simulation Is Shown on the Left; The Monitoring Layer called Interactive-Sim in Relation to the Simulation Executive and the Simulation Application Is Shown on the Right.

is layered between the simulation executive and the application simulation program and the **Clone-DB** database that is independent of the synchronization primitive of the simulation executive. From the point of view of the simulation executive, Interactive-Sim is a simulator application. In the context of the layered system Interactive-Sim is between the simulation executive and the user's application simulation program. Interactive-Sim itself is decomposed into sub-modules, where each is implemented for a particular simulation executive. The sub-modules are "pluggable"in that the appropriate submodule is plugged in for a specified simulation executive. New sub-modules can be implemented using a specified application interface. The structure, semantics and the functions that need to be implemented in a sub-module are described in a companion manual [Hybinette and Fujimoto 1999]. The general idea behind Interactive-Sim is that it is transparent to the simulation program, and also to the programmer utilizing the cloning primitives.

The key function of Interactive-Sim is to (1) intercept message sends and (2) to process events. For example, Interactive-Sim needs to know the message send and message receive primitives in order to intercept the invocation to process events or to forward copies of a message to cloned LPs. After interception, Interactive-Sim queries Clone-DB to determine message or process cloning. Interactive-Sim also intercepts functions that control or inquire about the number of LPs, since cloned simulation has a larger number of LPs than an un-cloned simulation. Interactive-Sim may require a minor adjustment of the interface between the simulator executive and its application to accomplish control of LPs, for example in GTW we protected accessibility to the constant `TWnlp` that returns the number of logical LPs in the simulation with a function that returns the same number (`int TWGetNumLPs()`). Similarly a function is used to set the constant: `TWSetNumLPs()` (See [Hybinette and Fujimoto 1999] for details).

The architecture of Clone-Sim is presented in Figure 5.5.

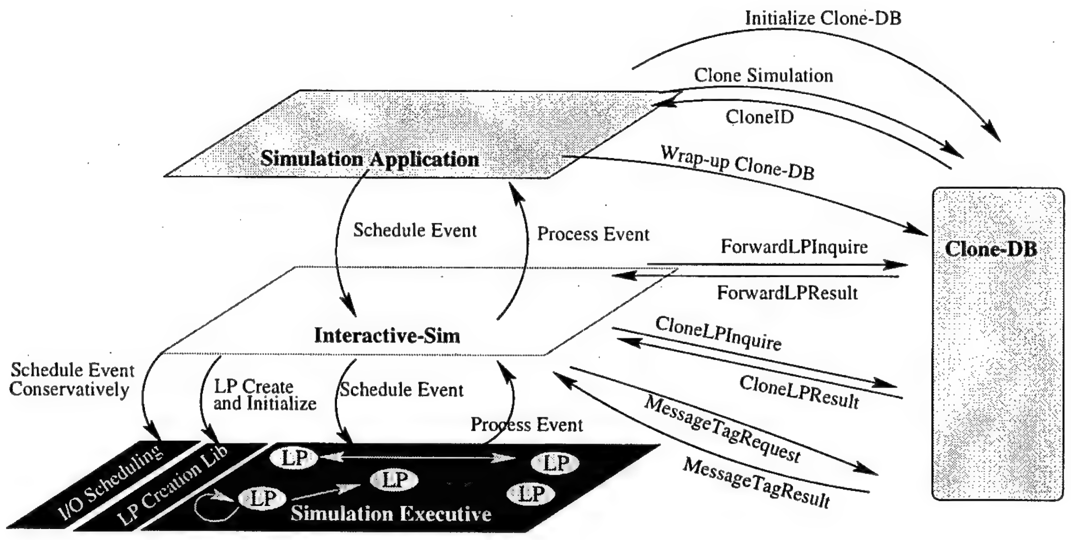


Figure 5.5: Clone-Sim Architecture and Application Programmer Interface

The next section will describe the simulation application programmer interface to the Cloning functions. These are the primitives that enables a programmer to clone simulations.

5.4 The Simulation Application

Clone-Sim accounts for the mapping of virtual LPs to physical LPs by assigning identifiers to each physical LP. There are three types of identifiers:

1. unique
2. global and
3. simulation.

The unique identifier (`UID_LP`) distinguishes physical logical processes. In addition each physical logical process is assigned to a global identifier (`GID_LP`), the same global identifier may be shared by multiple physical logical processes. The `GID_LP` corresponds to the original physical ID (before cloning) of the LP at time 0 (if this physical logical process is cloned the `GID_LP` may corresponds to multiple physical processes). Finally each version of a simulation (a version consists of a set of virtual LPs or a *plane* in Figure 5.1) is associated to a simulation identifier (`Clone_ID`). These identifiers can be accessed by specified functions that are described below.

There are currently six functions available to the simulation application. These are primitive functions, more complex cloning scenarios can be composed by these primitives. Each of these will be discussed below in detail. The API functions are:

```
void CloneSim_InitAppl( int argc, char ** argv )
void CloneSim_CloseAppl( void )
int  CloneSim_Create( int UID_LP, double current_sim_time )
int  CloneSim_Delete( int clone, double start, double end, CSFunc_p trig )
int  CloneSim_GetCloneID( int LP )
int  CloneSim_GID( void )
int  CloneSim_UID( void )
```

The utility of these functions depends on the state of the simulation. The state of the simulator is viewed as moving through three phases: initialization, execution of the simulation, and wrap-up. The initialization phase determines the initial number of logical processes, the mapping between processes and processors, and specifies event handlers. The simulation phase calls and schedulers event handlers specified for LPs. The wrap-up phase is where the simulation phase is complete and before the application terminates. Each of these phases are described in more detail below.

5.4.1 The Initialization Phase

The implementation assumes that the simulation executive allows the user application to set up the mapping. The mapping is thus exposed and allows Clone-Sim to manipulate the mapping. This is important, because Clone-Sim will exploit this ability and maintain the mapping transparently from the simulation application.

In order to initialize Clone-Sim, `void CloneSim_InitAppl()` is called at *the end* of the initialization phase. The function has two arguments: `argc` and `argv` that specify command line parameters that are passed to Clone-Sim. The main function of command line arguments is to specify the maximum number of clones that can be instantiated simultaneously. Details on these parameters are discussed in Section 5.5. Initializing Clone-Sim sets up data structures that: (1) control the mapping between logical processes and processors, (2) provide buffer space to cloned physical logical processes via a process buffer pool of LPs and (3) determine message or process cloning or determine child, sibling, parent relationships between cloned simulations.

In the current implementation, the mapping between logical processes and processors is assumed to be static after the initialization phase, however an extension is planned to allow the mapping to be dynamic and allow for automatic load-balancing. Also the maximum number of clones that can be instantiated simultaneously is specified during the initialization phase, currently this is specified via a command-line parameter (command-line parameters are defined in Section 5.5). An example initialization is shown below:

```
void InitializationPhase( int argc, char **argv )
  {
  /* code initialization is defined here */

  /* call initialization procedure for the cloning library */
  CloneSim_InitAppl( argc, argv );
  }
```

5.4.2 The Simulation Phase

During the simulation phase, cloning allows for the insertion and deletion of decision points via the cloning primitives `CloneSim_Create()` and `CloneSim_Delete()`. This can be implemented interactively or non-interactively by the simulation programmer. The event that causes cloning must be a **conservative** event (guaranteed to never rollback). If the decision point occurs on a set of LPs then a conservative event must be scheduled at the same simulation time by each of the LPs defined in the set of the same decision point.

The prototype of the function that allows for the insertion of a decision point is defined below:

`int CloneSim_Create(` int UID_LP, double current_sim_time)

The call returns an identification number of the newly cloned simulation, so that one can refer to the clone when deleting or pruning it. A negative number is returned upon error. The result is the instantiation of a new simulation. This represents the location in the execution path where the state of the newly created version start to diverge from the version that called it. As the function is called one new physical logical process is created. Any assignment to variables or calls to functions within this conservative event *after* the call to `CloneSim_Create()` *only effect the original clone.* Assignment or calls to functions within the conservative event before `CloneSim_Create()` effect both versions of the simulation: the newly created clone and the original clone.

The argument `UID_LP` is the unique identifier of the LP, and can be accessed via the call `CloneSim_UID()`. The argument current_sim_time is the simulation time of the event that calls the primitive. An example use of this function is included below:

```
void A_Conservative_Event( arguments )
  {
  int  unique_LP_identifier;
  int clone_identifier;

  /* simulator dependent code here */
  /* effects both original LP and instantiated LP below */

  /* access the unique logical process identifier of callee */
  unique_LP_identifier = CloneSim_UID();

  /* instantiates a new clone, a new logical process is created */
  clone_identifier
      = CloneSim_Create( unique_LP_identifier, current_sim_time );

  /* code here only effects caller LP of original simulation   */
  /* the new LP created via the Clone_SimCreate is un-effected */
  }
```

In addition to creating clones, Clone-Sim provides a mechanism to eliminate simulations that are not needed. This is done by installing a "trigger". The primitive which installs the trigger is: `CloneSim_Delete()`. The function can be called interactively or non-interactively. The prototype is:

`int CloneSim_Delete(` int clone, double start double end, CSFunc_p trigger)

The trigger is a condition defined by the argument **trigger** that is sampled within the simulation period specified by the arguments: **start**, **end**. The installation of the trigger only effects the logical process that installs the trigger and only the simulation whose version is given by the first argument: `clone`. So if all versions in the simulation need to be monitored the trigger needs to be installed for each version. If trigger is `NULL` the version that calls `CloneSim_Delete()` is pruned un-conditionally.

Currently, the pruning function only provides un-conditional pruning, conditional pruning is only available in an un-released version of Clone-Sim.

```
void Some_Event( arguments )
  {
  /* possibly some  simulator dependent code here */

  if( some condition )
     {
     /* prune if the simulation time of the callee is within the */
     /* simulation period [0.00, END_TIME] */
     CloneSim_Delete( cloneID, 0.00, END_TIME, NULL );
     }

  /* possibly some  simulator dependent code here */
  }
```

5.4.3 The Wrap-up Phase

When the simulation completes `CloneSim_Close( void )` should be called to clean up data structures and compute statistics. It should be called after the simulation code has completed and before terminating the program. The prototype is defined below:

```
int CloneSim_Close( void )
```

5.5 Command Line Parameters

The main purpose of the command line arguments is to set the maximum number of clones that can be instantiated simultaneously. The command line allows the programmer to set this parameter either implicitly by specifying the `cloning activation time` of each clone or directly by specifying the maximum number of clones. Cloning activation time is the earliest time (in simulated time) a clone can be scheduled.

The switch that sets the activation time is: `-c`. The switch is set for each clone that may be instantiated during the simulation. An example command line to enable the instantiation of a second clone at time 10 and a third at 20 is (the first clone is the initial simulation):

a.out -c 10 -c 20

When this option is used Clone-Sim parses the arguments and stores the times in a user-accessible array. Since clones may only be triggered from user code it is up to the application to monitor the simulation time and trigger clones appropriately.

The switch that sets the the maximum number of clones directly is: `-V` (where V is for versions). An example example command line that enables 10 simultaneously clones is:

a.out -V 2

Interactive-Sim initializes two data structures during command line processing:

- `CloneSim_CLONETIME` an array of doubles, and
- `CloneSim_NumClones` an integer.

Each element in `CloneSim_CLONETIME` corresponds to the activation time for a particular version that is instantiated, here index 0, refers to the first instantiated version, index 1 to the second instantiated version and so on. `CloneSim_NumClones` limits the number of clones that are instantiated, and is set directly by the switch `-V` or indirectly by the number of times the switch `-c` is set on the command line. If activation time is not set on the command line via the `-c` switch, then the activation time of each clone is set to the length of the simulation. For the `a.out` examples above `CloneSim_NumClones` is set to 2. The maximum number of clones in this case is three, where two clones are instantiated during the simulation.

The implementation cloning can set statically or dynamically. One way to implement dynamic cloning is to initially set the cloning time for each clone to the value that corresponds to the length of the simulation. To push the activation time for a particular version to an earlier time, its value can then be manipulated while the simulation is running. In the current release of Clone-Sim both `CloneSim_NumClones` and `CloneSim_CLONETIME` are accessible to the simulation programmer, in a later release however they will only be accessible via specified function primitives.

5.6 Compiling and using Clone-Sim

In order to utilize cloning the application needs to link with Clone-Sim via the flag `-lCloneSim`. The math library is also required, and is linked with the -lm flag. The Clone-Sim interface is defined in the header file: `cs_api.h` and must be included in each file that uses cloning primitives.

5.7 An Example

Clone-Sim consists of two modules: Clone-DB and Interactive-Sim. Clone-DB is independent of the synchronization mechanism of the simulator executive. A sub-module in Interactive-Sim in contrast must address the particulars of a simulator executive. There is a separate sub-module for each simulation executive supported. However, the programmer interface between Clone-Sim and the simulation application that is to be cloned is the same between all different simulation executives. Here, in this manual we assume that the sub-module has been implemented.

To illustrate the utility of Clone-Sim we will describe how cloning can be utilized for an application written for GTW using the cloning primitives. The same primitives may be used for other simulation executives, including simulator that utilizes a conservative synchronization mechanism. We assume that the reader has previous experience with writing applications for a simulator such as GTW, and we will provide high-level details as needed. Events in GTW is scheduled by specifying the destination LP and the time stamp increment. Each LP is instantiated by an event.

The GTW application that will be used for illustrative purposes is called P-Hold. P-Hold provides synthetic workloads using a fixed message population. The P-Hold simulations described here

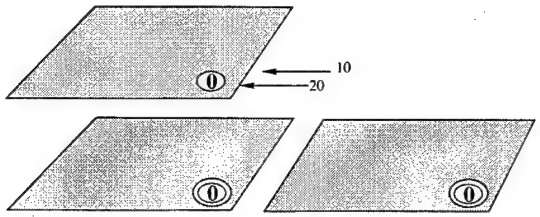


Figure 5.6: A Parallel Discrete Event Simulation Represented by Logical Processes (LPs) Replicated Twice (Where Each Plane Consists of a Set of LPs); the Upper Left Plane Shows the Original Simulation. (Only LP 0 is Shown In Each Version of the Simulation)

will use a message population of 256 (**message_population**) and 1024 logical processes (**number_lps**) for the length of the simulation (**END_TIME**) is set to 100 simulated seconds (for more details on P-Hold see [Fujimoto 1990b]). The message population is set by having each LP send **message_population/number_lps** messages during the initialization phase of the simulation (when simulated time is 0 for each of the logical LPs). For this application assume we want to instantiate two clones each from logical process 0: one clone is to be instantiated at simulated time 10 and the other at simulated time 20. A way to view this topology is shown in Figure 5.6, here LP 0 is physically cloned from the original simulation once at simulated time 10 and a second time at simulated time 20.

A GTW program consists of 3 phases: initialization, simulation and wrap-up. In addition a GTW program must define the structures for the state vectors of its logical processes and the message format to schedule events. In the below structure of the state space and each of the 3 phases of P-Hold is described.

GTW allows the user to specify the data types which give the executive an idea how to maintain the state space. Example data types include read-only, incremental and automatic. Figure 5.7 defines the state vectors and the message format for P-Hold events. The variables preceded by the prefix `cs_` are used to control cloning. There are two variables defined in the state space used for this purpose: `cs_CloneCount` and `cs_cloned`. `cs_CloneCount` is used to control the number of clones that are instantiated and `cs_cloned` is used to control the LP that instantiates the clone (in this example LP 0 will instantiate all clones). The message format in this case is simple and contains a single integer that counts the number of times a message "bounces" between LPs. An LP is instantiated by an event and a new event is scheduled by sending a message with a specified location LP and time, so in P-Hold each event corresponds to "a flow of messages", and the originator of the message flow can always be found by back-tracking from the receivers to senders up to simulated time 0.

The first phase of GTW is initialization. The initialization phase in GTW is defined by two procedures: `TWInitAppl()` and `IProc()`. `TWInitAppl()` defines global initialization and sets a number of things such as number of logical processes, event handlers, allocates the state space for the logical processes and so on. `IProc()` initializes the state of the LP and sends the initial messages to get the simulation started. The `IProc()` procedure always executes at simulated time zero. The specifics of `TWInitAppl()` and `IProc()` for the example are described in turn below.

The code `TWInitAppl()` for P-Hold is shown in Figure 5.7. In this example the simulation is homogeneous: each logical process defines the same initializing (`IPHoldLP()`), event handling (`PHoldLP()`) and finishing procedures `FPHoldLP()`. The LPs differ in having different random seeds (See the line that sets `Seeds[i]` in Figure 5.7). To initialize cloning `CloneSim_InitAppl()` is called at the *end* of `TWInitAppl()`. This enable Interactive-Sim to intercept the event handlers, and manipulate the format of messages.

`CloneSim_InitAppl()` takes two arguments: `argv` and `argc`. In the P-Hold example we set the cloning activation time[1] to 10 for the first clone and 20 for the second clone. This can be accomplished by passing: "`-c 10 -c 20`" via the above arguments. A GTW application sets several simulation parameters on the command line. Typically, number of processors and simulation time is set on the command line. In our example the command line to run P-Hold for 100 simulation seconds (set by the `-t switch`) using 4 processors (set by the `-p switch`) is:

```
phold -p 4 -t 100 -A
```

GTW uses the `-A` switch to allow command lines arguments to be passed to the `TWInitAppl()` procedure. All arguments following this flag are passed to the initialization procedure via `argc` and `argv`. This is shown above for illustrative purposes and the switch is not required if no arguments follow `-A`. In the example, the same arguments are sent to `CloneSim_InitAppl()` as in `TWInitAppl()`. An example P-Hold command line that enables the instantiation of a second clone at time 10 and a third at 20 (the first clone is the original simulation) is:

```
phold -p 4 -t 100 -A -c 10 -c 20
```

In the second part of the initialization phase GTW initializes the state space via the handler `IProc()` (set to `IPHoldLP()` in the example). `IProc()` can be viewed as an event at simulated time zero. `PHoldLP()` is shown in Figure 5.7. Here, `IPHoldLP()` sets the state vectors, specifies the state space that is automatically check-pointed and initializes the state variables. The state variable `cs_CloneCount` is set to 0, to indicate that currently the LP has not instantiated a clone and the state variable cs_cloned is set to 0 to enable cloning. To start the simulation each LP schedules **MsgPop/number_lps** messages, where the time stamp increment of each message is picked from a exponential distribution between 0 and 1; and the destination is picked from a uniform distribution consisting of all participating LPs.

[1] the cloning activation time is the earliest time a clone can be activated and is set by the switch `-c` (see Section 5.5 for more details).

After the application is initialized the simulation phase starts and the event handlers are called. The event handlers are specified in `TWInitAppl()`, and here all event handlers are set to `PHoldLP()`. The code for the event handler is shown in Figure 5.7. The procedure records the timestamp of the message that was just received and increments a counter indicating the number of messages received. The procedure determines cloning by evaluating (1) the activation time and the (2) identity of the logical process calling the procedure. The simulation clones LP 0 twice: once when LP 0 progresses beyond simulated time 10 and a second time when LP 0 after simulated time 20. The state variable `AVars.cs_cloned` is used to prevent undesirable clones. Children of logical process 0 are prevented to propagate further clones. A clone is instantiated by a conservative event that is defined by the procedure `ClonePHoldLP()` and is scheduled at simulated time TWNow() – resulting in a time stamp increment of zero, the first argument of `TWGetMsg()`. In this example, cloning is always instantiated by logical process 0. However, the primitives allows it to be instantiated by any logical process, even a logical process that has already been cloned.

The code for the event (`ClonePHoldLP()`) that instantiates cloning is shown in Figure 5.7. This event is scheduled conservatively from an event handler. In this case the event handler for the original clone of logical process 0, schedules the cloning event (the cloning event must be a conservative event to prevent rollbacks). During the conservative event the state variable `SV->AVars.cloned` serves to control the capability to clone further clones. If it is set to 0 then further cloning is allowed, if it is set to 1, cloning is disabled. The cloning event scheduled from `PHold()` prevents new clones from further cloning and preserves the right to the caller (the parent). The logical process that instantiates a simulation maintains the identification number of the clone it last instantiated, and its own identification number. Similarly, logical process that is cloned (the child) maintains the parent identification number, and its own identification number. This is also where the versions of the cloned simulations differ.

Before the simulation terminates Clone-Sim collects cloning statistics, and clears data structures. This is accomplish by calling `CloneSim_CloseAppl()` from the GTW function: `WrapUp_Appl()`. This code is shown in Figure 5.12. GTW wraps up applications by calling `FProc` for each logical process and then after finishing calling each finishing procedure for each LP it calls `WrapUp_Appl` once. The code for `FPHoldLP` is shown in Figure 5.13 and is an empty procedure.

```
#include "gtw.h"
#include <stdio.h>
#include <malloc.h>

/* remember timestamp on last MSZ messages */
#define MSZ 10

/* read only variables in state vector */
/* automatically check-pointed variables for LP */
struct MyAutoVars
        {
        TWSeed  Seeds;           /* seeds for random number generator */

        int     cs_CloneCount;   /* number of clones instantiated */
        int     cs_cloned;       /* controls who instantiated cloning */

        int     cs_parent;
        int     cs_child;
        };

/* incrementally check-pointed variables for LP */
struct MyIncVars
        {
        TWTime  LastTS[MSZ];     /* remember last MSZ timestamps */
        int     Count;           /* number of messages received */
        };

struct MyLPState
        {
        struct MyAutoVars  AVars;
        struct MyIncVars  ISVars;
        };

struct MyMsgData
        { /* Message data */
        int counter;
        };
```

Figure 5.7: Data structures that define state vectors and messages for P-Hold

```
void TWInitAppl( int argc, char **argv )
  {
  int  number_lps, i, message_population;

  number_lps         = 256;
  message_population = 1024;

  /* specify number of LPs */
  TWSetNumLPs( number_lps );
  for( i = 0; i < TWGetNumLPs(); i++ )
     {
     /* LP to PE mapping:  map round robin */
     TWLP[i].Map = i % TWGetNumGlobalPEs();
     /* set handlers: initialization, event and wrap-up functions */
     TWLP[i].IProc =  (FuncPtr2) IPHoldLP;
     TWLP[i].Proc  =  (FuncPtr)  PHoldLP;
     TWLP[i].FProc =  (FuncPtr2) FPHoldLP;
     TWLP[i].IncrSave = TRUE;
     /* set and allocate LP state */
     TWLP[i].State = TWMalloc( sizeof(struct MyLPState) );
     TWLP[i].LPStateSize = sizeof(struct MyLPState);
     TWLP[i].CopySize = sizeof(struct MyAutoVars);
     }
  /* get initial random number generator seeds */
  for( i = 0; i < TWGetNumLPs(); i++ ) TWRandInit( &(Seeds[i]), 0 );

  /* set memory mapping */
  for( i = 0; i < TWnpe ; i++ )
        for( j = 0; j < TWnpe; j++ )
           {
           if( i != TWnpe || j != i ) TWMemMap[i][j] = 1;
           if( i == j ) TWMemMap[i][j] = 1;
           }

  /* set message size */
  TWMsgSize = sizeof( struct MyMsgData );

  /* call initialization procedure for the cloning library */
  CloneSim_InitAppl( argc, argv );
  }
```

Figure 5.8: `TWInitAppl()` for P-Hold that uses cloning

```
void IPHoldLP( LPState *SV, int MyPE )
  {
  struct MyMsgData      *TWMsg;
  int                   i,
  TWTime                ts;
  TWEachSeed_t          s1, s2;
  int                   rcv_lp, dst_lp;

  struct LPState * CurState;
  CurState = GState[MyPE].CurState;

  myLP = TWMe();
  SV = (struct MyLPState *) TWLP[TWMe()].State;

  /* specify variables to be automatically check-pointed */
  TWAutoCheck( (char*)&(SV->AVars), sizeof(struct MyAutoVars) );

  TWRandGetSeeds( &(Seeds[TWMe()]), &s1, &s2 );
  TWRandSetSeeds( &(SV->AVars.Seeds), s1, s2 );

  /* initialize state variables */
  if( CurState->IncrSave )
      {
      SV->ISVars.Count = 0;
      for( i = 0; i < MSZ; i++ )
         {
         SV->ISVars.LastTS[i] = 0.0;
         }
      }

  SV->AVars.Count = 0;
  SV->AVars.CloneCount = 0;
  SV->AVars.cloned = 0;

  for( rcv_lp = TWMe(); rcv_lp < MsgPop; rcv_lp += TWGetNumLPs() )
     {
     ts = TWRandExponential( &(SV->AVars.Seeds), 1.0 );
     dst_lp = TWRandInteger( &(SV->AVars.Seeds), 0, TWGetNumLPs()-1 );
     TWGetMsg( ts, dst, sizeof(struct MyMsgData) );
     TWMsg->counter = 1;
     TWSend();
     }
  }
```

Figure 5.9: Procedure that initializes each LP for P-Hold

```
void PHoldLP( struct MyLPState *SV, struct MyMsgData *M, int MyPE )
  {
  struct MyMsgData     *TWMsg;
  struct LPState       *CurState;
  TWTime               ts;
  int                  dst;
  double               time_now;

  MyState  = &GState[MyPE];
  CurState = GState[MyPE].CurState;

  myLP = TWMe();
  time_now = TWNow();

  if( CurState->IncrSave )
      {
      TWCheckTWTime(&(SV->ISVars.LastTS[SV->ISVars.Count % MSZ]));
      SV->ISVars.LastTS[SV->AVars.Count % MSZ] = TWNow();
      TWCheck(&(SV->ISVars.Count));
      SV->ISVars.Count++;
      }
  SV->AVars.Count++;

  /* determine cloning scheduling */
  if( (TWMe() == 0) && (SV->AVars.cloned == 0)  )
      if( CloneSim_CLONETIME[SV->AVars.CloneCount] > 0.0 )
          {
          if( ((SV->AVars.CloneCount) < CloneSim_NumClones )
             && (CloneSim_CLONETIME[SV->AVars.CloneCount] < TWNow()) )
              {
              SV->AVars.CloneCount++;
              TWGetMsg( 0, TWMe(), sizeof(struct MyMsgData) );
              TWBlockingIOSend( (IOFuncPtr) ClonePHoldLP );
              }
          }
      }

  /* schedule a new event */
  ts  = TWRandExponential(&(SV->AVars.Seeds), 1.0);
  dst = TWRandInteger(&(SV->AVars.Seeds), 0, (TWGetNumLPs())-1);
  TWGetMsg( ts, dst, sizeof (struct MyMsgData) );
  TWMsg->counter = M->counter + 1;
  TWSend();
  }
```

Figure 5.10: Event handler procedure for P-Hold that may instantiate a clone

```
void ClonePHoldLP( struct MyLPState *SV, struct MyMsgData *M, int MyPE )
  {
  struct PEState       *MyState;
  struct LPState       *CurState;

  MyState  = &GState[MyPE];
  CurState = GState[MyPE].CurState;

  /* access the unique logical process identifier of callee */
  uid          = CloneSim_UID();

  /* disable this current clone and child clone to propagate */
  SV->AVars.cloned  = 1;

  /* instantiates a new clone, a new logical process is created */
  clone_identifier           = CloneSim_Create( uid, TWNow() );

  /* code here only effects caller LP of original simulation   */
  /* the new LP created via the Clone_SimCreate is un-effected */

  /* enable clone 0 to propagate more clones */
  SV->AVars.cloned  = 0;
  }
```

Figure 5.11: Conservative event procedure for P-Hold that instantiates a clone

```
void WrapUp_Appl( void )
  {
  CloneSim_CloseAppl();
  }
```

Figure 5.12: Wrap-Up procedure for P-Hold

```
void FPHoldLP( struct MyLPState * SV, int MyPE )
  {
  }
```

Figure 5.13: Finishing procedure for P-Hold

5.8 Reference Manual: Simulation Application

Details of the cloning functions available to the simulation application are described in below:

void CloneSim_InitAppl(int argc, char **argv)

- This procedure initializes the Clone-DB and sets up an LP buffer pool that is later used when a simulation is cloned. The addressing and handling of the LP buffer pool are transparent to the simulation application programmer.
- A list of arguments can be passed to Clone-Sim via the parameter: **argv_appl**. The number of arguments are given by the parameter **argc_appl**. The main-purpose of the command line arguments is to set the maximum number of clones that can be instantiated simultaneously. The command line allows the programmer to set this parameter either implicitly by specifying the `cloning activation time` of each clone (via the `-c` switch) or directly by specifying the maximum number of clones (via the `-V` switch). Cloning activation time is the earliest time (in simulated time) a clone can be scheduled.
- This procedure must be called at the end of the initialization phase before initializing each logical process.

void CloneSim_CloseAppl(void)

- This procedure clears data structures and collects cloning statistics.
- This procedure must be called when the simulation phase is complete and before the application terminates.

int CloneSim_Create(int UID_LP, double current_sim_time)

- This function creates a new simulation and clones an LP. It returns an identification number of the newly cloned simulation, so that one can refer to the clone when deleting or pruning it. A negative number is returned upon error. The invocation of **Clone_Create** can be viewed as the insertion of a decision point.
- The argument `UID_LP` is the unique identifier of the callee LP, and can be accessed via the call `CloneSim_UID()`. The argument current_sim_time is the simulation time of LP that calls the primitive.
- The event that calls this function must be a **conservative** event (A conservative event is an event that is guaranteed to never rollback). The argument: `current_sim_time` must be the same as the simulation time of the callee (the conservative event).
- If the a the decision points need to effect multiple LPs then each LP must schedule a conservative event that calls **CloneSim_Create()** where the argument: `current_sim_time` is equivalent. In this special case one version of the simulation is created and each of LPs that calls **CloneSim_Create()** is cloned.
- Within this event, the cloned and original simulation are similar up to the invocation of this function, all statements after the invocation are only applied to the original simulation.

int CloneSim_Delete(int cloneID, double start, double end, CSFunc_p func)

- This procedure prunes the cloned simulation identified `cloneID`. The pruning can depend on a trigger specified by a condition function `func()` and time period when to sample the condition specified by the argument: `func`, (`func` must return an integer).
- **cloneID** is a unique number specifying the clone that is pruned, the time period when the trigger `func` is affected is specified by the arguments `start` and `end`. The trigger is

specified by the argument: `func`, and (`func` must return an integer).

- **func** is a user defined function determining if a clone should be pruned. The clone is pruned if **func** returns 1. A **trigger** of NULL is equivalent to a return of TRUE and then the version of the simulation that calls `CloneSim_Delete()` is pruned un-conditionally.
- Currently, the pruning function only provides un-conditional pruning, conditional pruning is only available in an un-released version of Clone-Sim.

CloneSim_GetCloneID(void)

- This function returns the CloneID of the clone that invokes it.

CloneSim_GID(void)

- This function returns the corresponding logical process number of the original simulation (the first single un-cloned simulation). For example, if the simulation originally consisted of 2 logical processes numbered 0 and 1, then later, the simulation is cloned, all logical processes of the *cloned* simulation corresponding to logical process 0 and logical process 0 return a 0 when called from an event processed on logical process 0.

CloneSim_UID(void)

- This function returns a unique identification number of the logical process that invokes it.

Chapter 6

Cloning Parallel Simulations: Algorithm and Performance

The implementation of a cloning mechanism that allows for the evaluation of multiple simulated futures is presented and its performance is analyzed. A running parallel discrete event simulation is dynamically cloned at *decision points* to explore different execution paths concurrently. In this way what-if and alternative scenario analysis in gaming, tactical and strategic applications can be evaluated interactively or non-interactively. Performance results show that *virtual logical processes*, a new mechanism developed to avoid repeating common computations among clones improves efficiency.

6.1 Overview

The goal of this research is to develop mechanisms that support the realization of interactive simulation environments to enable rapid investigation of complex situations. In particular, the work describes here examines support to interactively explore different possible futures and compare their result by cloning a running parallel simulation. The tool supports critical decision making such as determining whether or not to mobilize reinforcements in a military scenario. Other applications that may benefit include gaming, strategic and tactical battle management and air-traffic scheduling.

A powerful simulation-based decision tool should provide for pausing, steering, monitoring, rollback and the evaluation of alternatives (a generalization of what-if scenario analysis). Pausing enables participants to interject new execution paths or to modify variables of the current execution path. Execution path modifications are added to the simulation as *decision points.* Steering is the insertion of decision points without pausing. Monitoring provides for querying and sampling simulation variables. These variables may be sampled continuously, triggered when some pre-defined condition is satisfied or presented on demand. Rollback provides for the re-evaluation of old decision points and variables. While the design and implementation support all these features, this chapter focuses on the details of the parallel evaluation of multiple alternatives.

Related work in interactive parallel discrete event simulation is described in [Steinman 1991] and in [Franks et al. 1997]. In [Steinman 1991] a message type called an *external* event enables interaction with the simulator. These events can steer, sample and query the simulation in progress. The approach of [Franks et al. 1997] allows for the testing of what-if scenarios provided they are interjected before a deadline. Alternatives are examined one after the other and the simulation must undo the effect of the previous alternative before considering another. In contrast to these methods, our algorithm dynamically creates and evaluates multiple alternatives concurrently using a cloning mechanism. New versions of a simulation in progress are cloned to evaluate alternatives. The alternative simulation proceeds in parallel with the original. Thus an increasing number of alternatives can be evaluated before resolution.

Parallel discrete event simulators provide speed up for time-consuming applications such as strategic battle-management, telecommunication networks, air-traffic scheduling and other large scale simulations. Strategic and tactical battlefield simulation in particular would benefit from our mechanism, since they often weigh options and must be responsive to unanticipated developments during a scenario. The interactive system is appropriate for both conservative and optimistic simulation protocols.

Cloning was suggested by von Neumann [von Neumann 1956] more than 40 years ago to provide fault-tolerance. Fault-tolerance is ensured by providing identical processes, identical transactions, duplicate data, or redundant services [Schneider 1990]. Cloning can also improve throughput by placing process replicas in the proximity of where a service is needed [Goldberg 1992a; Schneider 1990]. Cloning is also suggested as a solution for concurrency control in real-time databases [Bestavros 1994] and to improve accuracy of simulation results [Glasserman et al. 1996; Vaikili 1992; Glynn and Heidelberger 1991]. In the latter, the approach is to run multiple independent replications then average their results at the end of the runs.

The incremental update schemes of process migration algorithms such as [Zayas 1987a] are similar in philosophy to our virtual logical process scheme (covered in a later section). The common goal is to reduce the cost of copying the virtual address space between clones. The process migration algorithms differ in that only one active clone is provided for while we provide for multiple clones.

In this chapter we describe the implementation and performance of a simulation system that provides for the exploration of what-if and alternative scenarios. The goal of this research is to develop a model that supports an efficient, simple, and effective way to explore and compare alternate scenarios. The initial design of the cloning mechanism is described in [Hybinette and Fujimoto 1997]. Here we extend this work by describing the implementation and demonstrating its performance.

The chapter is organized as follows: The virtual logical paradigm is described in Section 6.2. This is followed by a discussion of the cloning mechanism in Section 6.3. We then describe the cloning mechanism implementation in Section 6.4. Next, performance measurements are presented and analyzed. We conclude with a summary and a discussion of future directions.

6.2 Virtual Logical Processes

A simple approach to provide for the exploration of computing multiple futures simultaneously is to replicate an entire simulation at a *decision point*, then independently execute each clone. Here, a decision point represents the location in the execution path where the state of the two versions start to diverge. One way to visualize this is to imagine the whole simulation domain replicated into many different planes where each plane is an independent version of the simulation executing in parallel with the other cloned versions (See Figure 6.1). However, this may not be suitable for large simulations because the cost of instantiating many objects (LPs) is prohibitively large. Furthermore, as the simulation progresses many computations may be duplicated between the planes even though

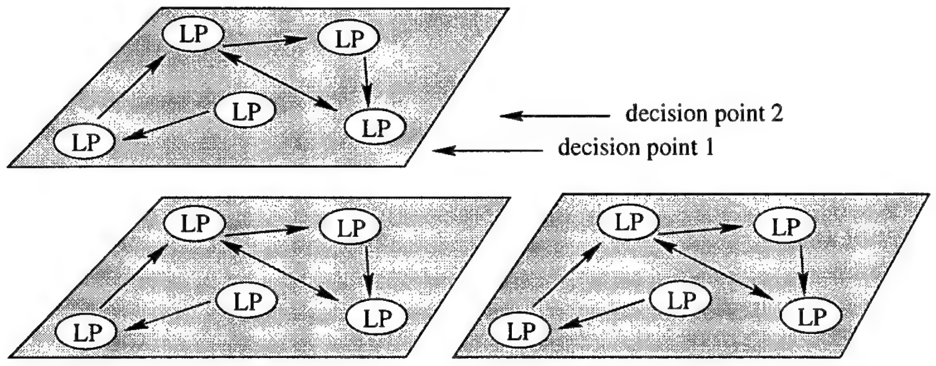


Figure 6.1: A Parallel Discrete Event Simulation Represented by Logical Processes (LPs) Replicated Twice; the Upper Left Plane Shows the Original Simulation

they could be shared.

To reduce the cost of computing multiple-futures the state space of each cloned plane is updated incrementally via the construct of *virtual logical processes.* In this paradigm each plane or *version* of the simulation contains a collection of virtual logical processes. A collection of virtual logical processes is created each time a new version of the simulation is cloned. The difference between the cloned simulations is in the mapping of virtual logical processes to physical logical processes. Here, a physical logical process refers to the run-time environment of virtual logical processes. Each virtual logical process (V) is assigned or *mapped* to a physical logical process (P). The idea is that virtual logical processes can share the same physical logical process thereby avoiding replication of common computations. But two physical processes cannot be mapped to the same virtual process. An analog is virtual memory where the same main memory address can shared by several virtual addresses, but two addresses in main memory cannot be mapped to the same virtual address. The mapping between virtual and physical processes is updated as the clones diverge. Resources are re-used as long as possible and only the smaller portions which cannot be shared are replicated.

In the remainder of the chapter the following notation is used:

- *version* i, the cloned simulation i.
- V, a virtual logical process.
- P, a physical logical process.
- V_j^i, the virtual logical process j version i.
- P_j^i, the physical logical process j version i.

The original simulation is version 1, the version number is incremented as simulations are cloned. The version number of a physical logical process is the same as the *lowest* version number of the virtual logical processes that maps to it.

Before a simulation is cloned there is one set of Vs and one set of Ps. A sample mapping of a simulation consisting of three logical processes before cloning is:

– V_A^1 maps to P_A^1
– V_B^1 maps to P_B^1
– V_C^1 maps to P_C^1

After a simulation is cloned a set of new physical processes are created for portions of the state that differ from the original simulation. The semantics of cloning is to instantiate the cloning *on* a set of physical logical processes called the clone set. The clone set may be specified dynamically. This

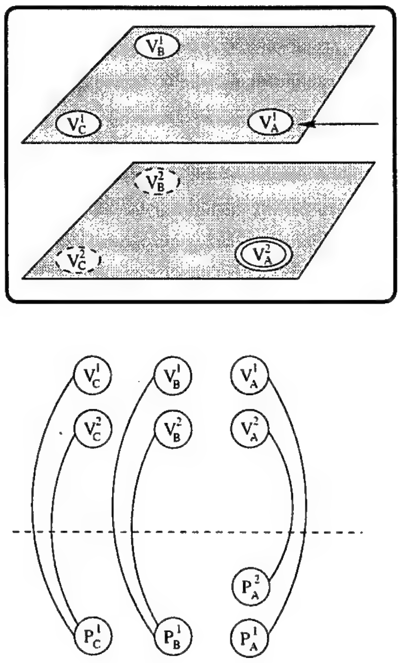


Figure 6.2: A Snapshot of a Simulation That Has Been Cloned; the Top Image Shows the Two Virtual Versions of the Simulation, the Bottom Image Shows the Mapping of the Virtual Processes to Physical Processes A, B and C

represents the insertion of a decision point and is where the state of the versions start to diverge. In addition to the new physical processes, a new set of virtual processes is created, resulting in a total of two sets of virtual logical processes. The first set or version of virtual processes are the same as the the original simulation before cloning, the second set represents a new version of the simulation that differs in its mapping to physical logical processes. The mapping of the new virtual logical processes created from cloning the simulation of three logical processes described above *on* process P_A^1 is:

- V_A^2 maps to P_A^2
- V_B^2 maps to P_B^1
- V_C^2 maps to P_C^1

The bottom image in Figure 6.2 shows the mapping between virtual processes and physical logical processes after the simulation is cloned on physical process A. Here the mapping of the original version of virtual processes stays the same and the computation between clones is shared. Virtual processes B and C, version one and version two share the same corresponding physical process; while virtual process A version one and version two maps to different physical processes. As the simulation progresses the mapping of virtual processes to physical processes changes, as new physical processes are created. Message sends and receives are carried out in the physical layer. In this manner a physical send corresponds to a *set* of sends in the virtual process layer. In the example (see Figure 6.3), a message sent from physical process B to physical process C implements two virtual message sends from virtual process B version 1 and version 2 to virtual process C versions 1 and 2 respectively.

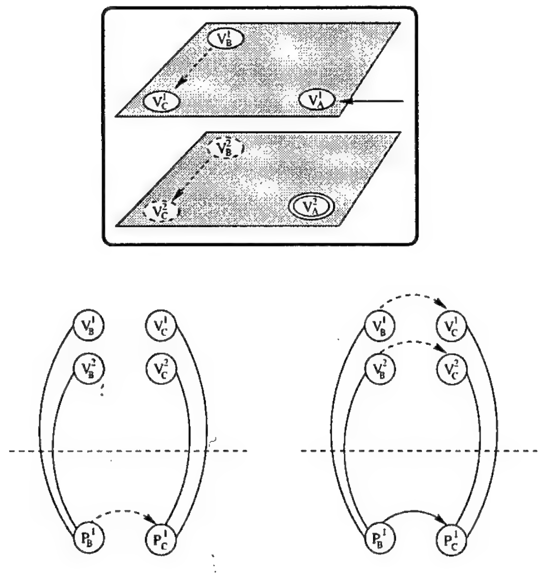


Figure 6.3: Transparent Send and Receive between Virtual Logical Processes

6.3 The Cloning Mechanism

As cloned versions of a parallel simulation progress physical processes may need to be cloned or messages originating from an un-cloned physical process may need to be forwarded. As mentioned earlier, a parallel discrete event simulator is effected by the communication of messages. Thus, the reception of a message is the point where a logical process considers message forwarding and process cloning. Here, two sets are inspected: the virtual processes mapped to the physical send process `VSendSet` and the physical receive process `VRcvSet`. The sender adds information to the message so that the receiver can construct the send set.

To illustrate the approach consider a military tactical simulation consisting of three objects: A theater command center represented by logical process B, a tactical wing represented by logical process C and a platoon represented by logical process A. The platoon in the original simulation (version 1) has half manpower while the cloned platoon (version 2) has full manpower.

6.3.1 Message Cloning

A physical logical process first determines if the receiver message must be cloned. A message is cloned when an un-cloned physical process sends a message to a physical process that has been cloned. In the context of the military scenario this happens when the theater command center (B) requests a status report from the platoon (A). Since, both versions of the command center are represented by the same physical process but there are two different physical processes that represent the platoon – two copies of the message are sent (See Figure 6.4). This can be determined by inspecting the send set and the receive set. The send set is $\{1, 2\}$ and the receive set is $\{1\}$, because virtual logical process version 1 and 2 map to the senders physical process and virtual logical process version 1 maps to the receiver physical process. Each physical process that maps to a version that is in the virtual send set receives a message. Since version 2 is in the send set and not in the receive set the physical process that received the message forwards the message to

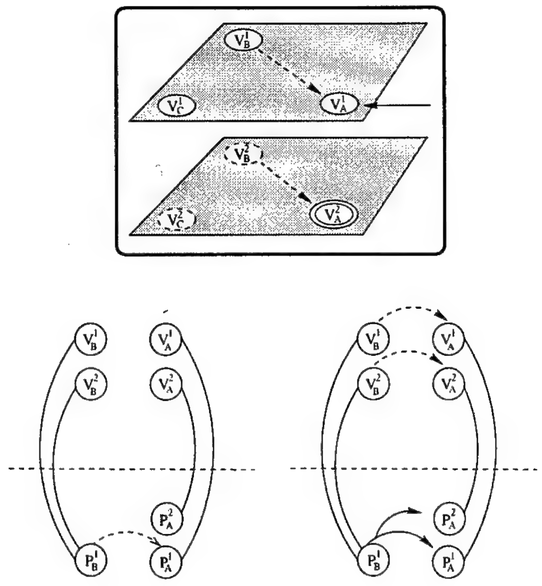


Figure 6.4: Multi-Casting a Message

physical process version 2. A process forwards a message to the lowest version that is not in the receive set. When a process forwards a message, the virtual processes that have seen the message (MsgSeenSet) is passed along as information in the message. This avoids redundant message sends. An algorithm that accomplishes these steps is listed below:

1. get VSendSet from message
2. get VRcvSet from local state
3. forwardSet = VSendSet $\cap$ $\overline{\texttt{VRcvSet}}$
4. forwardSet = forwardSet $\cap$ $\overline{\texttt{MsgSeenSet}}$
5. MsgSeenSet = MsgSeenSet $\cup$ VRcvSet
6. forward message to the virtual logical process with the lowest version number in the **forwardSet**

6.3.2 Process Cloning

After determining the physical processes that need to receive a message, the generation of a new physical process is considered. A physical process is cloned if there is a virtual logical process in the receive set that does not have a corresponding virtual sender or if the destination address of the message is a physical logical process that has not yet been created. For example, the process representing the theater command center (P_B^1) clones itself in the military example if the half-staffed platoon (P_A^1) sends it a message to request extra manpower (See Figure 6.5). This corresponds to the sending of a physical message from P_A^1 to P_B^1. Since virtual process version 2 of processes B should not receive the message, version 2 of the virtual receivers should be prevented from being influenced by this message. This is achieved by changing the mapping between virtual and physical processes. The result is that the process is cloned *before* an event is processed so that the state of the physical process is not influenced by the incoming message. The new "version 2" of physical process of B is prevented from the misconception that A needs extra manpower. In this scenario the send set is $\{1\}$ because the set contains version 1 of virtual senders mapped to physical process

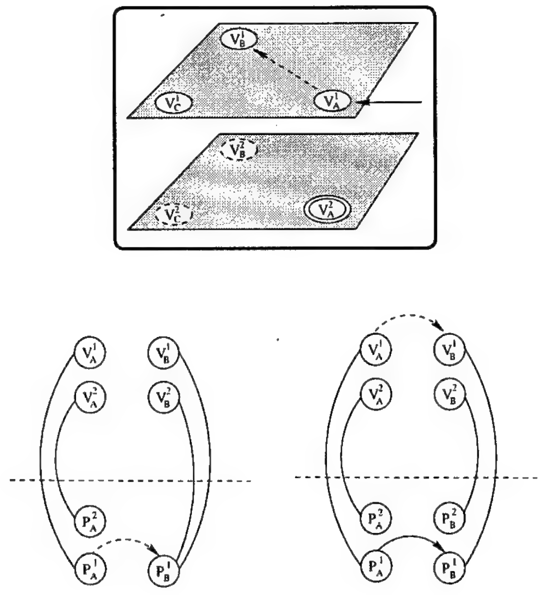


Figure 6.5: Creation of a Physical Logical Process

P_A^1. The receive set is $\{1, 2\}$ since both version 1 and version two maps to the physical receiver P_B^1. The algorithm that determines process cloning is listed below:

```
1. get VSendSet from message
2. get VRcvSet from local state
3. VPsUnaffected = $\overline{\texttt{VSendSet}} \cap$ VRcvSet
4. if ( VPsUnaffected <> $\emptyset$ ) then
       Clone a Physical LP and assign it the same version number as the lowest virtual LP in
       the set VPsUnaffected
```

6.4 The Optimistic Protocol

The optimistic implementation uses a rollback function to keep LP mapping information consistent. One implication is that if a recipient does not exist, the parent accepts the event, then clones the LP and then sends an instantiation message to activate a child. As a result, the original destination field in the event is overwritten with the parent's address. Before the destination is overwritten, however, it is saved within the message. During rollback the original destination and mapping information is restored via the rollback function.

There are two main data structures used for this procedure, a relationship table and a mapping table. The relationship table keeps track of the shape of the planes i.e., which planes cloned which other planes. This structure is modified when a simulation is cloned (not when a LP is cloned). The number of instantiated planes is only limited by memory. The data structure is read by other LPs to determine its children upon the receipt of a message.

The second data structure, the mapping table, is maintained by the active LPs. During instantiation of a clone the parent initializes the index that belongs to its child. Thereafter, the child itself maintains its own information. This data structure contains the mapping information, birth time stamps and whether the LP is active or not. To minimize mapping updates at the interjection of a decision point an LP keep track of which children are physically cloned. The physical LP can then compute its mapped virtual LPs. A sending LP piggy-backs this on a message so the receiver can compare its mapping to determine cloning or forwarding. The mapping table is referenced to determine whom to clone and to whom to forward messages. It is also referenced to determine which LP to rollback or to determine which LP should handle the message when a message is addressed to an LP that does not exist before the time stamp of the message.

6.5 Cloning Evaluation

The performance benefit of cloning can be realized on two levels: (1) simulations that are interactively cloned at a decision point share the same execution path before cloning (i.e. less computations), and (2) after cloning the versions of the simulation can share computations and messages via the virtual logical processes construct. Since a multitude of dynamically created versions of a simulation are encompassed within a single simulation engine, the approach is especially suited for highly interactive environments that require complex scenario analysis, such as gaming, interactive steering and what-if scenario analysis.

Cloning achieves efficiency by intercepting the communication primitives of a simulator executive. By monitoring the send and receive primitives Cloning avoids unnecessary duplication of logical processes (LPs). Likewise, it can determine which logical processes need copies of messages. The key idea is that the receiving LP can determine whether to clone or forward by inspecting bits that are piggy-backed by the cloning mechanism on messages.

The evaluation of dynamic cloning is made in the context of three applications: one is a a synthetic load benchmark called P-Hold, the the second is called PCS, a benchmark which simulates a personal communication services network and the third is a air-traffic simulator called the Detailed Policy Assessment Tool or DPAT. These first two applications are described in the next two sections. Then their performance are evaluated. In order to generalize the applicability of cloning we also evaluates its performance in the context of a commericial decision tool called the Detailed Policy Assessment Tool (DPAT). Here, we are investigating the performance of typical usage scenarios and these results are also contrasted with the performance of P-Hold with the same type of usage scenarios.

6.5.1 P-Hold

P-Hold provides synthetic workloads using a fixed message population. Each LP is instantiated by an event. Upon instantiation, the LP schedules a new event with a specified time-stamp increment and destination. The destination LP is specified within the message as well. The P-Hold simulations use a message population of 8096 and 1024 logical processes (for more details on P-Hold see [Fujimoto 1990b]).

6.5.2 PCS

Another set of experiments use PCS, a benchmark which simulates a personal communication services network. The network is wireless (radio receivers and transmitters) and provides services to mobile PCS subscribers. The radio ports are structured in a grid with one port per sector. Each cell services *portables* or mobile phone units that each occupy a cell for a period of time before proceeding to one of the four neighboring cells. Call arrival, call completion, and move are example behaviors or types of portables which are modeled by events in the simulation. Cells are modeled by LPs. Here we use a PCS network of 1024 cells (a $32X32$ grid) and 8192 portables. Most of the communication, typically 90%, is between LPs residing on the same processor. Many of those messages are self-initiating, or sent between the same logical process. More details of PCS are described in [Carothers et al. 1994].

Experimental Results of P-Hold and PCS

Evaluation of the system for the below experimanets was conducted on an SGI Power Challenge with twelve 75 MHz MIPS R8000 processors. The first level instruction and data caches are each 16 KB. The unified secondary cache is 4 MB. The main memory size of the power challenge is 3 GB. All experiments used four processors. The data collected for each plot are typical values of a series of at least 15 runs. Two benchmarks are used to measure performance: P-Hold and PCS.

To illustrate the impact of incremental cloning, the performance of replication (two physically replicated simulations) and a single cloned simulation are compared with the incremental cloning scheme. The replicated simulation immediately physically replicates all logical processes. The single clone simulation is a lower bound of cloning (one clone implies that the original simulation has not been cloned). In all cases the decision point occurs at one logical process (LP_0) at simulated time 100. Simulated time is also referred as the virtual time to distinguish it from wall-clock time.

Cloning is evaluated with P-Hold by varying the rate at which a clone causes the cloning of other LPs. This is done by adjusting the selection of the destination LP of a message (logical processes are instantiated by receiving messages). In the first case the spreading is slowed by having an LP send messages only to itself or to its neighbor (the local distribution). In the second case the spreading is not constrained and the destination is selected from a uniform distribution of all logical processes in the simulated system. The time stamp increment to schedule a new event is 1.0 in the local case and selected from an exponential distribution between 0.0 and 1.0 in the uniform case.

Figure 6.6 shows simulation progress (virtual time) of P-Hold in the local case versus wall clock time. Figure 6.7 shows the performance of P-Hold where the destination address is selected from a uniform distribution. In both plots a larger number indicates better performance.

The local case shows that the simulation initially slows down upon the instantiation of cloning, then continues linearly. The uniform case shows a more dramatic slow-down but then proceeds linearly as well. The local cloned case performs significantly faster than the uniform case, once the performance stabilizes. The uniform case progresses about 25 seconds faster than the replicated case, and the local case is about 75 seconds faster. This is a significant difference for an interactive simulation, where alternatives need to be compared and instantiated dynamically. As one would expect, the plots indicate that cloning is more advantageous for simulations using local interactions.

To illustrate how cloning spreads to other logical processes a second metric is evaluated, namely the ratio of messages that are not shared between physical processes with the same version number. The ratio r is given by V_1/V_0, where V_1 and V_0 is the number of events computed by the newly cloned LP and the original LP respectively. A result of $r = 0$ implies the cloned LP have performed no event computations (all computations have been shared between cloned LP and its parent) and $r = 1$ implies that the cloned LP and the original LP have performed the same number of

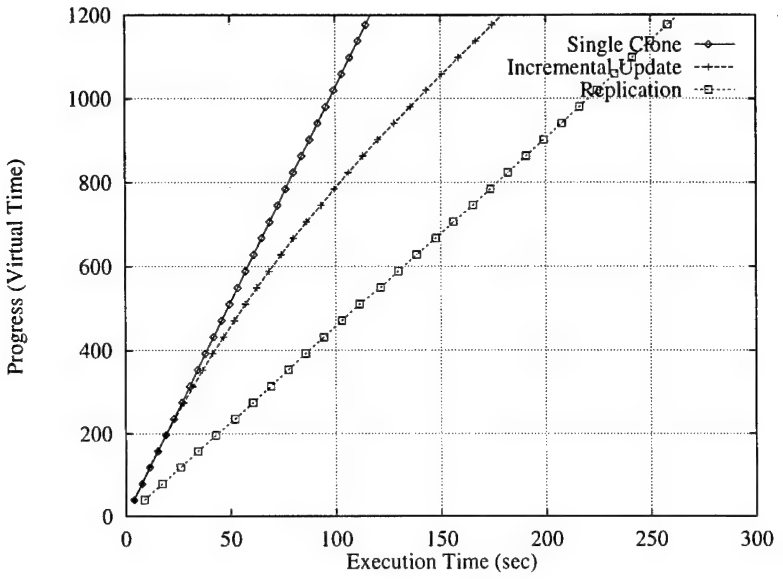


Figure 6.6: Performance of P-Hold Where Destination LP Is Either Self or Nearest Neighbor. Larger Numbers Indicate Better Performance

computations. A nonzero value r indicates that the LP has been cloned.

The plot of V_1/V_0 show how cloning spreads to other logical processes over a simulation period. Figure 6.8 shows the event ratio of non-shared computations for P-Hold in the uniform case and Figure 6.9 shows it for the local case. Each line in the graph is a snapshot at a specific virtual time of the distributed event ratio of all logical processes within the simulation. The uniform P-Hold clones all its logical processes within the first 6 simulated time units after cloning (the spreading occurs between cloning at time 100 and the completion of cloning of all processes at simulated time 106). The local distribution takes more than 1100 simulated time units to clone all its logical processes.

The change in linear progress observed in Figures 6.7 and 6.6 reflects the spreading of the cloning. The faster the spreading the quicker the slow-down. In the local case the slope is constant after cloning has spread to close to three quarters of the logical processes, or at 900 simulated time units. Similarly, the slope in the uniform case is constant 6 time units after the instantiation of the initial clone which is also when over three quarters of the processes have been cloned. Figure 6.10 shows the plot of a simulation that magnifies the region between simulated time 90 and 110 of Figure 6.7, here the slow-down after the initial cloning at time 100 can be observed. After cloning has completed the slopes of both the replicated and the incremental scheme is half of the single clone simulation after all the logical processes have been cloned in both the local and uniform cases. This is not surprising since in the former there are two simulations as opposed to one in the latter.

The performance of cloning using the PCS benchmark is shown in Figure 6.11. PCS behaves similar to local P-Hold initially, but then degrades toward the replicated PCS simulation. At wall clock time 150 second the replicated simulation surpasses the cloned simulation in performance. One factor impacting performance is a change in the rollback rate. Figure 6.13 shows rate of roll-backs as the simulation progresses. The graph shows that PCS's rollback rate suddenly drops after the initiation of the cloning event then rapidly increases until it reaches simulated time 700,000 to stabilize. PCS may become unstable because the cloned LPs starts inducing additional roll-backs

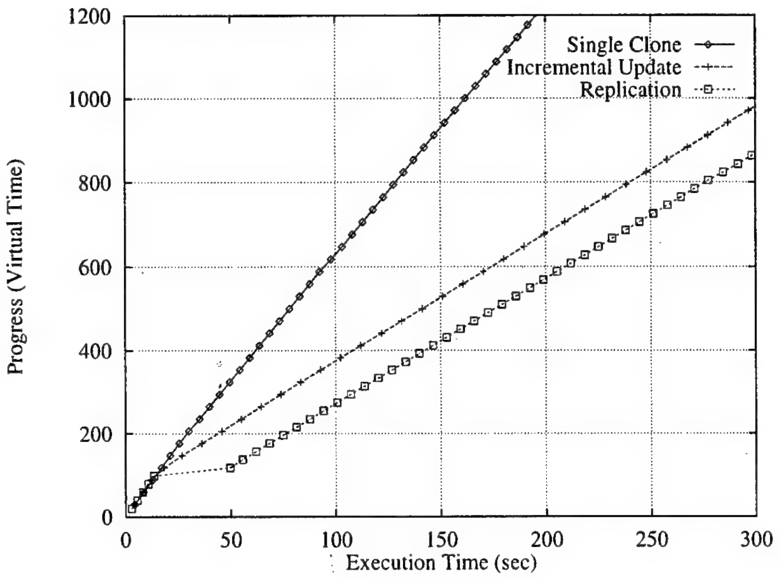


Figure 6.7: Performance of P-Hold Where Destination LP Is Selected from a Uniform Distribution of All LPs in the Simulation. Larger Numbers Indicate Better Performance

in un-cloned LPs. This change is dramatic because the rollback rate is typically low, and a small change makes a significant impact. For example, in both of the single cloned cases of P-Hold the rollback rate is over 1000 roll-backs per second, but in the PCS the rollback rate is below 150 roll-backs per second. This behavior, and development of countermeasures are currently under investigation.

These observations suggest that cloning with incremental update may be advantageous for applications that have a low spreading factor and a moderate to high rollback rate. It may be beneficial for application that require the analysis of many alternatives.

6.5.3 The Detailed Policy Assessment Tool (DPAT)

In order to generalize the applicability of cloning we expand on the above work that evaluates the cloning mechanism using P-Hold (a synthetic benchmark) and PCS (a realistic benchmark that simulates personal communication services networks). To investigate the utility of "cloning" for real world tasks we visited the MITRE corporation in Virginia to investigate in typical usage scenarios of a commercial decision tool called the Detailed Policy Assessment Tool (DPAT).

DPAT is an air traffic control application that computes congestion related delays and throughputs in the air traffic control system. The software can simulate different parts of international airspace. i.e Asia, Taiwan, the continental US and more. The system models the behavior of each airport and the airspace between them. The airspace is divided up into sectors. Each sector can enforce restrictions such as number of aircraft occupying a sector at specific time. Two control restrictions effect the flow of aircraft: Ground-delay programs and Miles-in-trail (MIT).

Departing aircraft are regulated by Ground-Delay-Programs. Ground delay programs prevent aircraft from departing an airport, in order to avoid circling in the air which burns fuel unnecessary. En-route traffic is regulated by restricting the miles-in-trail which is the minimum distance between aircraft. The allowed minimum distance differs depending on weather conditions. In good weather

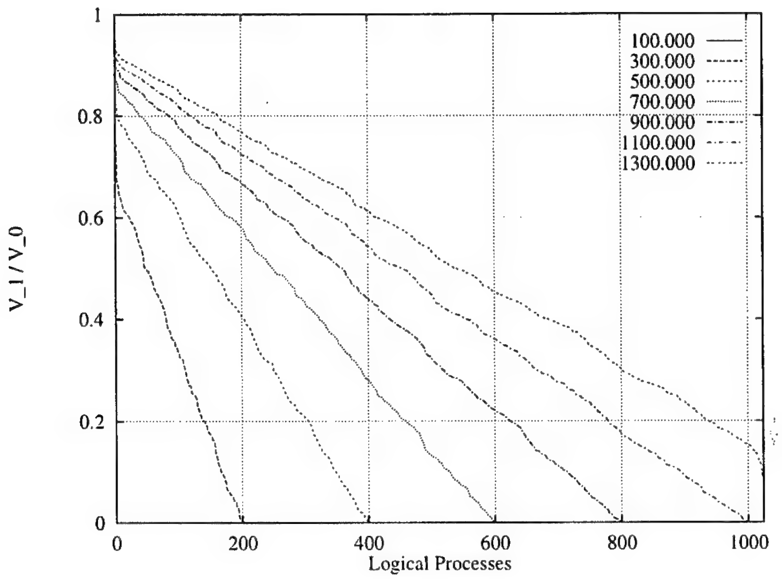


Figure 6.8: Event Computation Ratio Distribution of P-Hold Dominated by Local Communications. Lower Values Indicate More Shared Computations

the minimum distance allowed is 2 miles, while bad weather (decreased visibility) the minimum distance is increased to 5 miles for greater safety. The FAA can evaluate the effect on traffic flow by manipulating the MIT restrictions in simulation. For example, the FAA may want to see if they can increase safety by increasing MIT without impacting the traffic flow rate. The remaining discussion will focus on evaluating the effect of MIT restrictions.

With dynamic cloning an increased number of alternative MIT restrictions can be evaluated in time-constrained situations. Dynamic cloning can be used to effectively address two particular questions regarding MIT restrictions: *What* MIT value is the most beneficial and *when* should a new MIT value be enforced? These questions suggest the two types of usage scenarios depicted in Figure 6.14. On the left, alternative values of MIT are injected dynamically into a running simulation as soon as congestion is anticipated at an airport. On the right, different simulated time values determining when to enforce the restriction are evaluated. These proposed approaches differ from the current approach where MIT is a global value with respect to a running simulation. Different MIT values and their initiations are simulated independently and run to their completion then their result are evaluated. If un-anticipated MIT values are needed more simulations are run.

Experimental Results of Typical Usage Scenarios

Evaluation of the system was conducted on an SGI Origin 2000 with sixteen 195 MHz MIPS R10,000 processors. On this machine, the first level instruction and data caches are each 32 KB. The unified secondary cache is 4 MB. The main memory size is 4 GB. All experiments used four processors. Each data point represent the mean of series of at least 5 runs. Two applications are used to measure performance: DPAT and P-Hold.

DPAT is described in the previous section. P-Hold provides synthetic workloads using a fixed message population. Each LP is instantiated by an event. Upon instantiation, the LP schedules a new event with a specified time-stamp increment and destination. The destination LP is specified

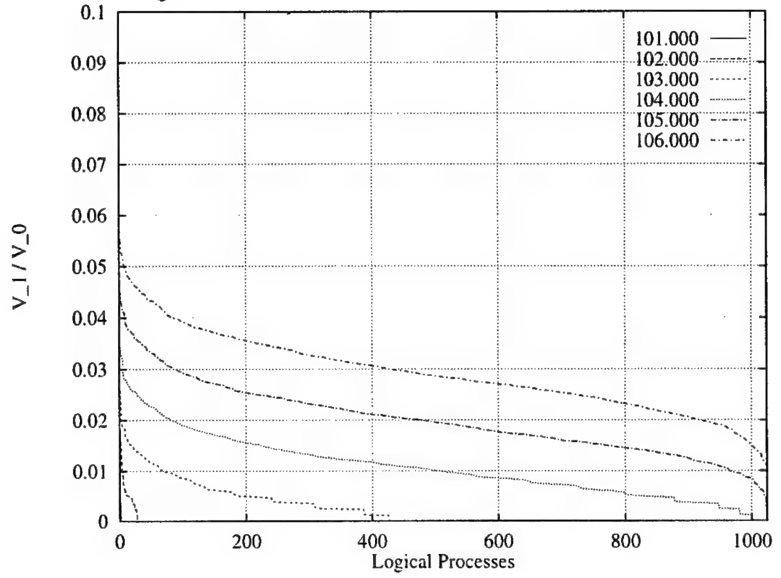


Figure 6.9: Event Computation Ratio Distribution of P-Hold With Uniformly Distributed Communications. Lower Values Indicate More Shared Computations

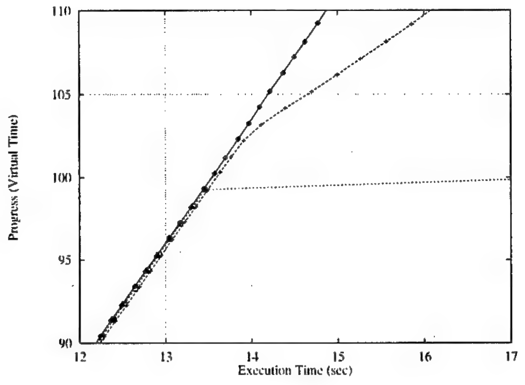


Figure 6.10: Local P-Hold Performance in the Region between Simulated Time 90 and 110

within the message as well. The P-Hold simulations use a message population of 8096 and 1024 logical processes (for more details on P-Hold see [Fujimoto 1990c].

To illustrate the impact of dynamic cloning, the performance of a traditional batch-processing scheme, where each simulation path is run separately one after the other is compared with the dynamic cloning scheme. Performance is speedup of the total time of using cloning relative to batch processing.

The independent variables are the inception of the cloning point (in simulated time) and the number of clones. Simulated time is also referred to as *virtual time* to distinguish it from wall-clock time.

The plot on the left in Figure 6.15 shows the performance of DPAT when several clones are generated on different MIT values at the same simulation time. The plot on the right shows the performance of DPAT when one MIT value is evaluated at several initiation times (for reference the performance of a single clone is shown as well). The results indicate that dynamic cloning always

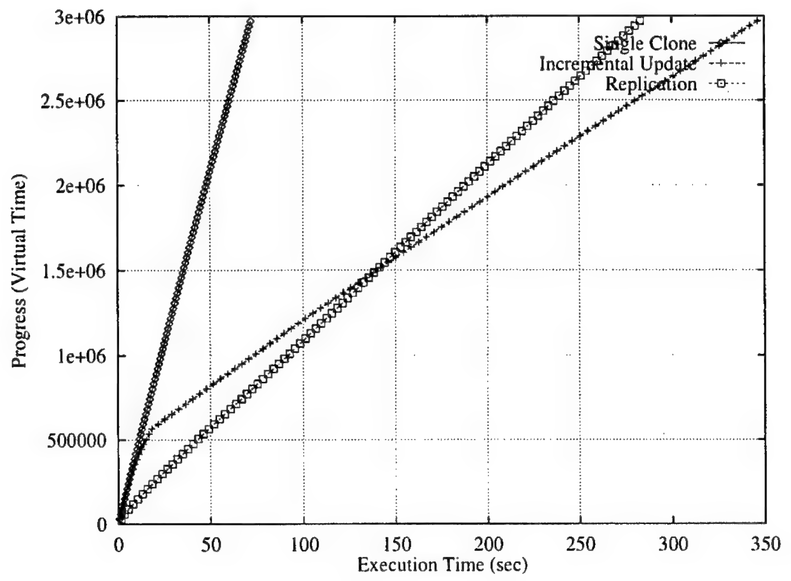


Figure 6.11: Performance of PCS. Larger Numbers Indicate Better Performance

outperforms the traditional batch scheme approach when running multiple clones. Dynamic cloning becomes more advantageous as the number of alternatives increases or the later the decision point is inserted. The plots show that the dynamic cloning scheme can run 240 percent faster than the batch scheme approach (e.g. for simulating different MIT values: instantiating 5 clones at 2,500 simulated minutes takes 75.38 seconds for the dynamic schema versus 183.60 seconds for the batch scheme).

Cloning is evaluated with P-Hold by varying the rate at which a clone causes the cloning of other LPs. This is done by adjusting the selection of the destination LP of a message (logical processes are instantiated by receiving messages). In the first case the spreading is slowed by having an LP send messages only to itself or to its neighbor (the local distribution). In the second case the spreading is not constrained and the destination is selected from a uniform distribution of all logical processes in the simulated system. The time stamp increment to schedule a new event is 1.0 in the local case and selected from an exponential distribution between 0.0 and 1.0 in the uniform case. In the cases of instantiated multiple clones, each new clone is instantiated with the same time stamp.

The plot on the left in Figure 6.16 shows the performance gain of P-Hold in the local case over batch processing (when running multiple clones, the results of a single clone is shown for reference). The plot on the right in Figure 6.16 shows the performance gain of P-Hold in the uniform case over batch processing.

In the local case dynamic cloning always outperforms the traditional batch scheme approach. In the uniform case however, the batch scheme performs better if cloning is initiated early and when evaluating more than 3 alternatives. This is probably because the spreading is quick, and as a result very few computations are shared between the clones. If the cloning event occur later, more computations are shared and again dynamic cloning outperforms batch processing. As in DPAT, the advantage becomes more dramatic the later the impetus of the decision point or the larger the number of alternatives. For the local case dynamic cloning improves performance by as much as 313 percent and in the local case by as much as 450 percent (e.g. instantiating 6 clones at virtual time 550 takes 53.04 seconds for the dynamic schema versus 239.27 seconds for the batch

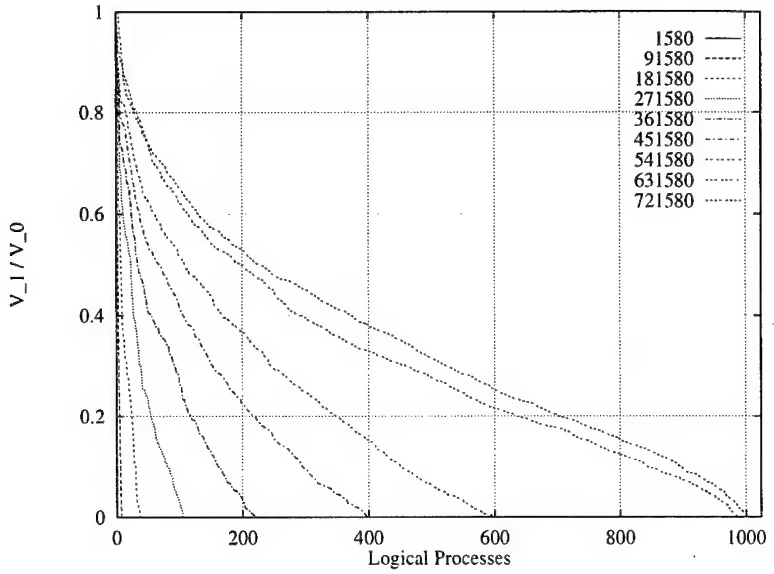


Figure 6.12: Distribution of Event Computation Ratio of PCS. Lower Values Indicate More Shared Computations

processing scheme). As one would expect, the plots indicate that cloning is more advantageous for simulations using local interactions. The dramatic improvement over the batch-processing is due to the incremental up-dates scheme. The performance gain the local case highlights that incremental updating further improves the benefit of dynamic cloning.

6.5.4 Pruning Clones

In addition to cloning, pruning is evaluated. Pruning is activated when a clone is determined not to provide useful result. Pruning, can be interactive or activated via a trigger defined at the insertion of the cloning point. To illustrate the acceleration of pruning, the performance can be observed by measuring the progress of virtual time as a function on real time. Here, simulation evaluates approximately 2 days of air-traffic delays or 3000 simulated minutes. Cloning occurs at simulated time 804 and pruning is initiated by a trigger that is activated at simulated time 1,116.

The plot on the right in Figure 6.17 shows three curves: a single run on the left, a cloned run (2 clones) that is pruned in middle and a cloned simulation that is pruned on the right. The acceleration of pruning, or the change in slope, is immediately noticeable at the activation of the pruning trigger (at simulated time 1,116). The pruning curve instantly conforms to the slope of the single curve.

In a similar manner to the heuristic to estimate the performance of cloning, the benefit (time saved) of pruning can be predicted by multiplying three factors: the proportion after pruning, the number of prunes and single execution time:

$$\begin{aligned}\text{Estimated Time Benefit} &= \text{single proportion} * \text{single time} \\ &+ \text{multiple proportion} * \#\ \text{prunes} * \text{single time}\end{aligned}$$

Pruning is assumed to occur after diffusion (see illustration on left in Figure 6.17). As an illustrative example consider the performance curves described above. Here, the simulation completes

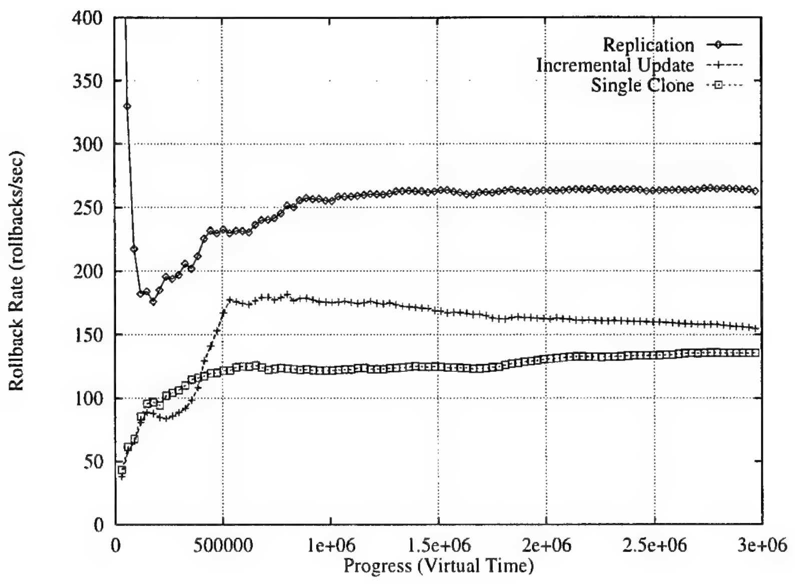


Figure 6.13: Rollback Rate of PCS

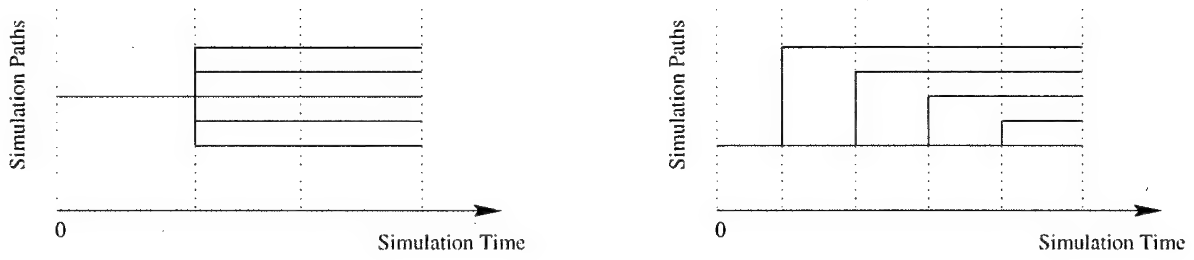


Figure 6.14: DPAT usage scenarios. Left: several clones generated on several MIT values at the same simulation time. Right: one MIT value is evaluated at several initiation times. Branch points are shown as solid vertical lines

at simulated time 3,000. Each curve complete in real time at 15.93 seconds for the single cloned run, 20.33 for the cloned and pruned run, and 30.30 seconds for the cloned run (note, that pruning improves the performance by 33 percent over not pruning the simulation). In this example the the pruned heuristic predicts that the performance improvement should be 9.73 seconds:

$$\begin{aligned}\text{Estimate Time Benefit} &= \frac{3,000 - 1166.70}{3,000} * 15.93 \\ &= \mathbf{9.73} \text{ seconds}\end{aligned}$$

The actual performance benefit is:

$$\begin{aligned}\text{Actual Time Benefit} &= 30.30 - 20.33 \\ &= \mathbf{9.97} \text{ seconds}\end{aligned}$$

This estimate is only 2.4 % off the actual performance, and is thus a good predictor of performance. The implementation allows pruning to occur during the diffusion phase, even though the evaluation heuristic assumes otherwise.

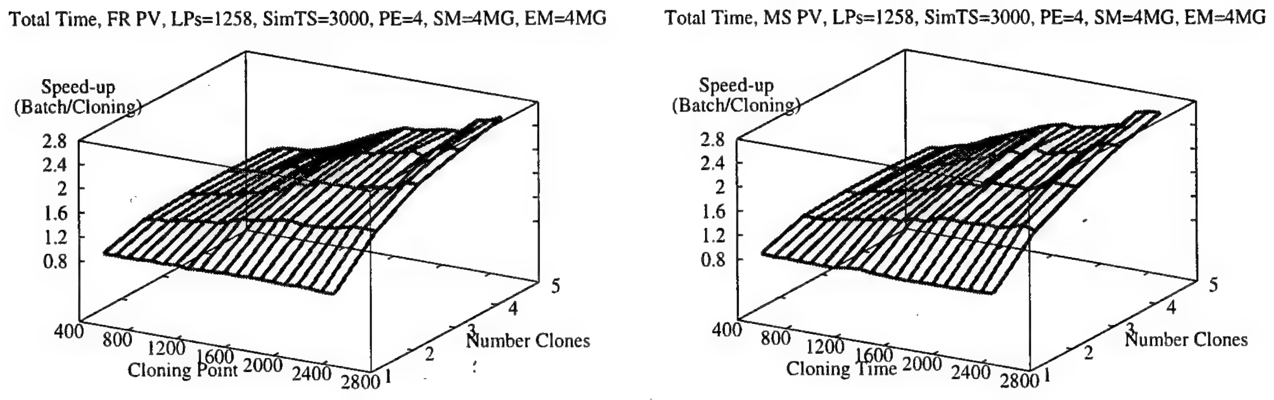


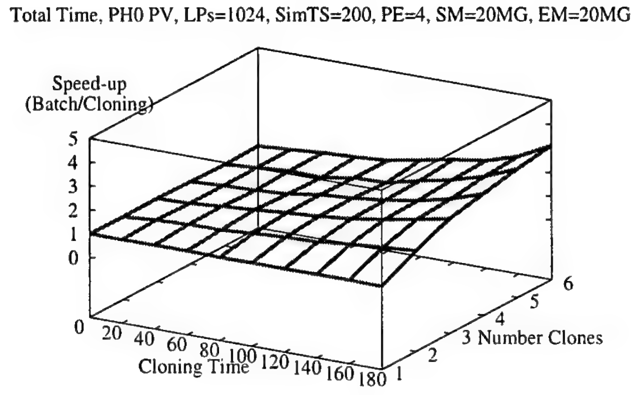
Figure 6.15: Dynamic cloning significantly improves the performance of DPAT over batch processing. Left: speedup of cloning over batch processing for DPAT when several clones are generated on different MIT values at the same simulation time. Right: speedup of DPAT when one MIT value is evaluated at several initiation times. Larger Numbers Indicate Better Performance

6.6 Conclusions and Future Work

Dynamic cloning allows for the exploration of different possible futures in interactive parallel simulation environments. The mechanism is applicable to conservative and optimistic simulation protocols. Before the insertion of a decision points clones share the same execution path and thus avoid un-necessary computations. To avoid copying the *entire* state upon instantiating a clone, the state is cloned incrementally. This is done by assigning the state of a virtual logical process to a physical logical process.

Performance results suggest that the scheme is especially efficient when the influence of a message introduced by a logical process does not spread to other processes in the simulation at a high rate. The results also show that the scheme is beneficial for applications that need to explore many different alternatives, because they can take advantage of the cumulative time-saving of each clone. The results also show the benefit of pruning.

Currently, it is assumed that a user can interject decision points into the simulator. Defining a mechanism for automating the interjection of decision points is a topic for future research. This may be accomplished by balancing the "running ahead" and "branching" depending on the probability of particular branches against available computational resources and real time constraints. This may offer an effective tool to explore the ordering of simultaneous events [Wieland 1997] and in the analysis by stochastic methods. Another area of research is to use cloning as a mechanism for collaborative work. Here, challenges include how different clones can be shared between different users and how different users can merge their solutions with other clones.



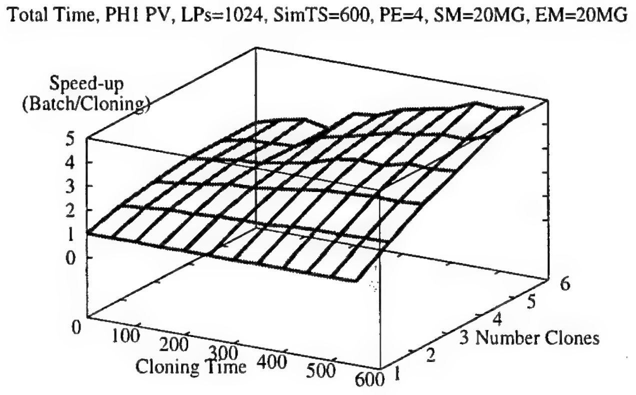
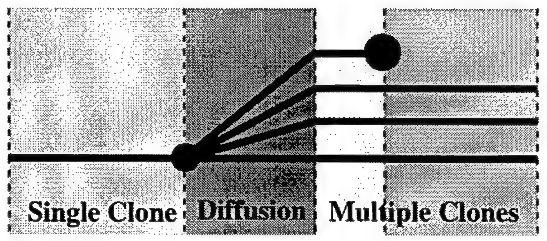


Figure 6.16: Incremental updating improves the performance of P-Hold over batch processing. Several clones are generated simultaneously at the same simulation time. Left: speedup of cloning over batch processing where destination LP is selected from a uniform distribution of all LPs in the simulation (fast spreading). Right: speedup of where Destination LP is either self or nearest neighbor (slower spreading). Larger Numbers Indicate Better Performance



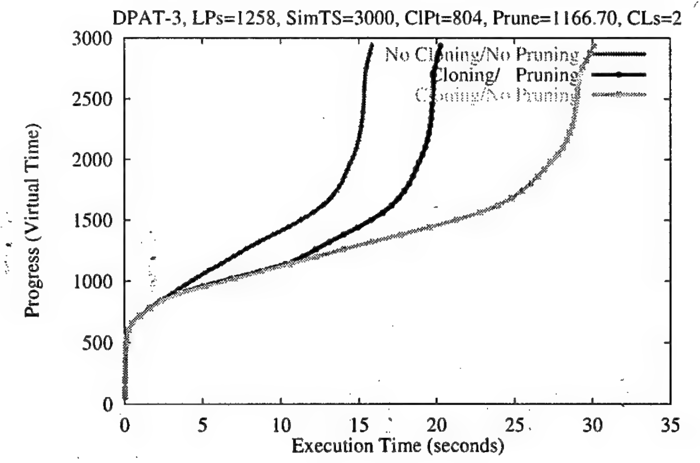


Figure 6.17: Pruning of DPAT

Chapter 7

Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms

Time Warp is an optimistic protocol for synchronizing parallel discrete-event simulations. To achieve performance in a multi-user network of workstation (NOW) environment, Time Warp must continue to operate efficiently in the presence of external workloads caused by other users, processor heterogeneity and irregular internal workloads caused by the simulation model. However, these performance problems can cause a Time Warp program to become grossly unbalanced, resulting in slower execution. The key observation asserted in this chapter is that each of these performance problems, while different in source, has a similar manifestation. For a Time Warp program to be balanced, the amount of wall-clock time necessary to advance an LP one unit of simulation time should be about the same for all LPs. Using this observation, we devise a single algorithm that mitigates these performance problems and enables the "background" execution of Time Warp programs on heterogeneous distributed computing platforms in the presence of external as well as irregular internal workloads.

7.1 Introduction

Time Warp is an optimistic synchronization mechanism develop by Jefferson and Sowizral [Jefferson and Sowizral 1982] used in the parallelization of discrete-event simulation. The distributed simulator consists of a collection of *logical processes* or LPs, each modeling a distinct component of the system being modeled, e.g., a server in a queuing network. LPs communicate by exchanging timestamped event messages, e.g., denoting the arrival of a new job at that server.

The Time Warp mechanism uses a detection-and-recovery protocol to synchronize the computation. Any time an LP determines that it has processed events out of timestamp order, it "rolls back" those events, and re-executes them. For a detailed discussion of Time Warp as well as other parallel simulation protocols we refer the reader to [Fujimoto 1990a].

With few exceptions, most research on Time Warp to date assumes the simulation program has allocated a fixed number of processors when execution begins, and has exclusive access to these processors throughout the lifetime of the simulation. Specifically, interference from other, external computations is minimal, and no provisions for adding or removing processors during the execution of the simulation are made. In fact, in most experimental studies, one typically goes to great lengths to eliminate any unwanted external interference from other user and system computations in order to obtain performance measurements that are not perturbed by external workloads. These extreme measures are taken because a Time Warp program that is well-balanced when executed on dedicated hardware may become grossly unbalanced when executed on machines with external computations from other users. Logical processes (LPs) that are mapped to heavily-utilized processors will advance very slowly through simulated time relative to others executing on lightly loaded processors. This can cause some LPs to advance too far ahead into the simulated future, resulting in very long or frequent rollbacks. While good from an experimental standpoint, this "dedicated platform" paradigm is often not the prevalent paradigm one encounters in practice. In particular, networks of desktop workstations (NOWs) and distributed compute servers consisting of collections of workstation-class CPUs interconnected through high-speed LANs have become prevalent. Despite the continual, reduced cost of computing hardware, shared use of computer resources will continue to be a common computing paradigm in the foreseeable future.

Moreover, this multi-user computing environment is typically composed of heterogeneous workstations that contain different processors. These processors may have quite different performance characteristics that, if not taken into consideration, can lead to poor Time Warp performance.

In addition to performance-robbing external workloads and processor heterogeneity, the application model itself can be a source of perturbation. In many simulation models, the amount of CPU time required to process an event varies among logical processes (LPs), and the event population may differ across LPs, both of which can degrade performance.

The key observation asserted in this chapter is that each of these performance problems, while different in source, has a similar manifestation. For a Time Warp program to be balanced, the amount of wall-clock time (i.e., elapsed real-time) necessary to advance an LP one unit of simulation time should be about the same for all LPs. Using this observation, we devise a single algorithm that mitigates these performance problems and enables the "background" execution of Time Warp programs on heterogeneous distributed computing platforms in the presence of external as well as irregular internal workloads.

To demonstrate the effectiveness of our *Background Execution (BGE)* algorithm, we construct an experimental testbed. In particular, we develop a model of a wireless (PCS) communications systems that can be executed atop our Georgia Tech Time Warp (GTW) system [Fujimoto et al. 1997]. What is special about this application is that it is an instance of a class of applications that are *self-initiated* [Nicol 1991]. Here, LPs typically schedule most of their events to themselves, which leads to relatively few remote messages making this class of applications well suited for the NOW platforms, which are known to have high remote communication overheads.

Using this simulation model in addition to a synthetic benchmark application, we demonstrate our Background Execution (BGE) algorithm is able to: (i) dynamically allocate additional CPUs during the execution of the distributed simulation as they become available and migrate portions of the distributed simulation workload onto these machines, (ii) dynamically release certain CPUs during the simulation as they become loaded with other, external, computations, and off-load the workload to the remaining CPUs used by the distributed simulation, and (iii) dynamically redistribute the workload on the existing set of processors as some become more heavily or lightly loaded by changing, externally or internally induced workloads.

The remainder of this study is organized as follows: Section 7.2 characterizes the different kinds of workload imbalances Time Warp programs can be subjected. Section 7.3 presents related work.

Section 7.4 presents our BGE algorithm that is suitable for balancing the load in the presence of irregular internal workloads, and external workloads. We then describe the implementation of our BGE algorithm in Section 7.5. The benchmark applications used in this experimental study are discussed in Section 7.6. Results from an experimental performance study are then presented where we compare the performance of our GTW system with and without our BGE algorithm in Sections 7.7-7.10. Section 7.11 summarizes our results and presents future research directions.

7.2 Characterization of Time Warp Workload Imbalance

For Time Warp programs, there are three sources of internal workload imbalance: (i) *event-population*, (ii) *event-granularity*, and (iii) *communications.* With irregular event-population workloads, the number of events processed over a period of simulated time may differ among LPs. Consequently, some LPs induce a much greater "load" on the processor than others because they have to process more events to reach the same point in simulated time. This can have a detrimental impact on Time Warp performance, resulting in the under-loaded processors becoming "overly-optimistic" and being rolled back by the over-loaded processors. Applications that suffer from this behavior include digital logic circuit simulations [Briner 1991; Soule and Gupta 1991] and National Airspace System models [Wieland et al. 1995; Wilson and Nicol 1996] and PCS models where the portable population differs among calling areas.

A special case of an unbalanced event-population workload, is an unbalanced *LP-population* where the number of LPs per processor differs significantly. We make LP-population a special-case because the driving force behind the simulation is the event-population. That is to say if we add LPs to a simulation where all LPs are the same without adding any additional events, the forward execution costs will remain approximately the same.

Load imbalances caused by irregular event-granularity workloads occur when the amount of wall-clock-time to process an event varies greatly among LPs. We assume that state-saving overheads are part of event processing costs. Thus, the event-population for each LP could be the same, but the amount of time a processor takes to process the events varies. As with event-population workloads, these kinds of irregular workloads also cause the over-loaded processors to constantly rollback the under-loaded ones, leading to poor Time Warp performance. Applications that exhibit this kind of internal workload include network simulations, such as SS7 [Xiao and Gomes 1993], ATM [Hao et al. 1996] as well as battle manager simulations, such as TISES [Office 1995]. PCS models, such as those presented in [Carothers et al. 1995], can also be configured to generate event-granularity workload imbalances.

The last cause of internal workload imbalance is communication. Here, an LP or group of LPs change their communication pattern so that new off processor communications are introduced that did not exist previously. The consequence of this is that some set of processors are now using more CPU cycles to send and receive remote messages, which can be quite costly in a NOW environment. The effect on Time Warp performance is that the communication-laiden processors act as if they are loaded and slow their rate of advance through simulated time. Meanwhile the other processors advancing at a much faster rate will be rolled back, thus reducing system performance. These communications workloads typically happen in applications where the "action" is subject to radical changes, such as the Eagle combat model [Rich and Michelsen 1991]. Due implementation-specific limitations, our BGE algorithm is currently unable to mitigate this type of workload imbalance.

We do however suggest some solutions in Section 7.11.

In addition to irregular internal workloads, Time Warp programs executing in a multi-user, NOW environment can be subjected to two sources of external workload imbalance. The first source is a *user-induced* external workload, which occurs when a user or group of users executes a program either locally or remotely on the same pool of computing resources that is currently being used by the Time Warp program. These external workloads effectively "steal" CPU cycles from the Time Warp program, assuming the operating system is "fair" in its allocation of CPU resources, causing some processors to become "over-loaded" with work. This can seriously degrade the performance of the Time Warp program. Here, LPs mapped to the "over-loaded" processors will require more wall-clock time to advance through simulated time, allowing "under-loaded" processors to advance at a much faster rate. These faster processors become "overly optimistic" and are consistently rolled back as the slow processors send them straggler events that arrive in an LPs past, resulting in a "thrashing" rollback behavior and degraded Time Warp performance.

Included in user-induced workloads are *system-induced* workloads. These are workloads generated by the network operating system, such as automated backup and software distribution programs, such as `depot`.

The last source of external workload imbalance is *processor heterogeneity*, where processor performance characteristics vary among workstations in a NOW. From a Time Warp programs point of view, workstations containing older, slower processors appear as though they are over-burdened with work when compared with more advanced workstations. As with user-induced workloads, heterogeneous processors create a situation where LPs mapped to slow processors progress at a rate less than that of LPs mapped to faster processors, resulting in long rollbacks because the faster processors become "overly optimistic", greatly reducing the performance gains of the distributed simulation.

With the exception of the communication workload imbalance, we will demonstrate how our BGE algorithm is able to detect and mitigate each of these workload imbalances.

7.3 Related Work

Background execution is essentially a load management problem. Traditionally, load balancing or load sharing involves distributing the workload across a fixed set of processors in order to minimize the elapsed time to execute the program. Many load balancing techniques designed to support distributed systems (e.g., NOWs) have been proposed and implemented, such as [Willebeek-LeMair and Reeve 1993]. However, when executing Time Warp programs on a NOW, the traditional load balancing problem must be extended in three ways which necessitates the need for a new algorithm. First, the load balancing mechanism must take into account dynamically changing external workloads produced by other computations. These external loads cannot be controlled by the load management software. Second, the set of processors that can be utilized by the distributed simulation expands and contracts during the execution of the program in unpredictable ways. Finally, the set of processors may have different performance characteristics that must be taken into consideration.

Optimistic synchronization mechanisms introduce new wrinkles to dynamic load management: high processor utilization does not necessarily imply good performance because a processor may be busy executing work that is later undone. Further, there is a close relationship between load management and the efficiency (e.g., amount of rolled back computation) of the synchronization mechanism, as discussed earlier. These factors necessitate development of load management techniques specific to Time Warp. Thus, process/object migration systems such as Accent [Zayas 1987b], Amoeba [Mullender et al. 1990], Charlotte [Artsy and Finkel 1989], Condor [Litzkow and Livny 1990], Locus [Thiel 1991], MOSIX [Barak et al. 1993], MpPVM [Chanchio and Sun 1996], RHODOS [Zhu and Goscinski 1990], Sprite [Ousterhout et al. 1988], and V-System [Theimer et al. 1985] that distribute jobs onto networked workstations as independent processes (i.e., not parallel program processes) to "soak up" otherwise unused CPU cycles, are not sufficient for Time Warp simulations.

Dynamic load management of Time Warp programs has been studied by others. Reiher and Jefferson propose a new metric called *effective processor utilization* which is defined as the fraction of the time during which a processor is executing computations that are eventually committed [Reiher and Jefferson 1990]. Based on this metric, they propose a strategy that migrates processes from processors with high effective utilization to those with low utilization. Reiher and Jefferson also propose splitting a logical process into *phases* to reduce the amount of process state that must be moved when an LP migrates from one processor to another. Glazer and Tropper propose allocating virtual time-slices to processes, based on their observed rate of advancing the local simulation clock [Glazer and Tropper 1993]. They present simulation results illustrating this approach yields better performance than the Reiher/Jefferson scheme for certain workloads. To our knowledge, this scheme has not been implemented on an operational Time Warp system. Goldberg describes an interesting approach to load distribution that replicates bottleneck processes to enable concurrent execution [Goldberg 1992b]. Time Warp is used to maintain consistency among the replicated copies.

More recently, Wilson and Nicol [Wilson and Nicol 1996] devise a method for automated load balancing in the SPEEDES parallel simulation environment [Steinman 1992]. The scheme they propose collects computation data, which consist of the number of events processed by each LP. These data are saved in a file and used during subsequent runs to statically partition the simulation application onto the set of available processors. Using this approach, the performance of the current run is only improved based on data collected from previous runs of the simulation.

Additionally, Avril and Tropper [Avril and Tropper 1996] present a scheme for dynamically load balancing Time Warp programs with irregular, internal workloads. Here, *processor load* is defined to be the number of events which were processed by the LPs assigned to that processor, including events rolled back and re-processed. Using this scheme, they improve Time Warp's throughput by 40 to 100% for VLSI models from the ISCS'89 benchmarks, where throughput is defined to be the number of non rolled-back events per unit of time. A fundamental difference between this approach and ours is the explicit exclusion of virtual time in the load balancing metric. The consequence of this decision is that it implicitly assumes that all events span the same amount of virtual time. Moreover, by using event counts in the processor load metric, it assumes that all events have the same computational requirements. While these two assumptions are true for VLSI simulation models, it is not true of all simulation models, such as TISES [Office 1995]. Consequently, these assumptions limit the utility of their approach.

None of these approaches address the question of balancing the load in the presence of external workloads and processor heterogeneity. The approach proposed here utilizes the ideas of not considering rolled back computation in deriving load balancing metrics, and workload allocation based on the rate of simulated time advance in developing an approach for background execution.

Burdorf and Marti [Burdorf and Marti 1993] propose an approach to periodically compute the average and standard deviation of all the LP local clocks in the system. If the average local clock among the LPs mapped to a processor is greater than the system-wide average plus one standard deviation, it is concluded that this processor is advancing too rapidly through simulated time, so additional LPs are migrated to that processor to "slow it down." Specifically, the LP with the smallest local clock is moved. In addition, other LPs that have low virtual clocks (specifically, a local clock less than the system-wide average minus one standard deviation) are moved to the processor that has the LP with the largest local clock. Marti and Burdorf observe that this approach will balance the workload in the presence of external computations competing for the same processors. A drawback with this approach is that it depends on virtual time differences among LPs to detect load balances. *Message-initiated* applications [Nicol 1991] can exhibit behaviors which allow under loaded processors to advance their LPs only to be rolled back later due to late arriving messages caused by slow processors. Thus, when the load distribution algorithm takes a snapshot of where LPs are with respect to virtual time, it could be that all LPs are at the same point, despite the presence of a load imbalance. Moreover, throttling techniques, such as RiskFree TWOS [Bellenot 1993], SRADS [Dickens and Reynolds, Jr. 1990], Adaptive Flow Control [Panesar and Fujimoto 1997], Elastic Time Algorithm [Srinivasan and Reynolds, Jr. 1995] and Breathing Time Warp [Steinman 1993], limit the advance of processors such that LPs are not allowed to become "overly optimistic" and may all at the same point in virtual time, despite an obvious workload imbalance. Consequently, under certain application workloads or when this approach is combined with other risk-limiting, throttling mechanisms, load imbalances may go undetected.

Schlagenhaft et al. [SchlagenHaft et al. 1995] propose an approach to balance the load of a VLSI circuit application on a distributed Time Warp simulator in the presence of external workloads. They define an inverse measure of the load, called *Virtual Time Progress*, which reflects how fast a simulation process continues in virtual time. This approach has some similarities with ours, however neither it nor Burdorf's approach address the question of dynamically changing the set of processors utilized by the simulation ,processor heterogeneity, or dynamically changing, internal workloads.

Work in dynamic load balancing for conservative parallel simulations has been done as well. Most recently, Boukerche and Das [Boukerche and Das 1997] devise a novel load balancing scheme for an optimized version of the Chandy-Misra null message algorithm [Chandy and Misra 1979]. Their approach introduces the notion of CPU-queue length as measure of workload on each processor. This workload measure is determined for both real message and null messages on each processors and combined using a weighted average function to compute the overall workload on a processor. Using this approach they reduce synchronization overheads in the Chandy-Misra algorithm by 30-40% when compared to the use of a static load balancing algorithm. While good results are achieved with this approach for conservative simulation protocols, it is not appropriate for optimistic protocols, because an unknown amount of work that is currently pending on a processor *could be* later undone. Optimistic protocols, such as Time Warp, require a measure of workload that only considers the amount of *committed* computation.

7.4 The BGE Workload Management Policy

The load management policy used here consists of two components:

1. The *processor allocation* policy that defines the set of processors that may be used by the Time Warp program. In general, this *usable set* of processors will change dynamically throughout

the execution of the distributed simulation.

2. The *load balancing* policy that migrates LPs between processors in the usable set. This policy must maintain efficient execution, in spite of dynamically changing external workloads in the processors in the usable set. It is assumed the Time Warp system has no control over these workloads, nor control over priority of execution in the operating system of these external computations relative to the Time Warp program.

Dynamic load distribution of individual LPs burdens simulations containing large numbers (say, thousands) of LPs. This is because a large number of entities must be considered by the load balancing algorithm, increasing the computation required for load distribution, and load balancing information that must be maintained. Further, because migrating each LP requires a certain amount of overhead, independent of the "size" of the LP, migrating many LPs from one processor to another is less efficient than migrating a group of LPs as a single unit. Here, LPs are first grouped (by the application) into *clusters* of LPs, and the cluster forms the atomic unit that can be migrated from one processor to another. In addition to reducing load management and process migration overheads, this approach will keep LPs that frequently communicate together, on the same processor, provided they are grouped within the same cluster. We assume the modeler has enough knowledge of the application to do a good job of "clustering" the LPs together. Once an LP is assigned to cluster it remains so for the life time of the simulation.

In our BGE algorithm, a central process is responsible for monitoring the processors that are to be used by the distributed simulation. This process executes periodically at a user defined *scheduling interval*, denoted by $T_{schedule}$ and estimates the expected amount of CPU time that would be allocated to a Time Warp simulation if it were to execute on that host, based on the current workload of this host over the last schedule interval. The load balancing policy is responsible for assigning LP clusters to processors. Our implementation of the load balancing policy uses a central process that executes periodically every $T_{schedule}$ seconds to determine which clusters should be moved to another processor or processing element (PE).

7.4.1 Processor Allocation

Initially, all processors are in the usable set. However, as shown in Figure 7.1, should a processor lose all of its clusters as a result of normal load balancing, the BGE algorithm records the current load and removes that processor from the usable set. We call this load the *DeAllocLoad.* This processor remains unusable until the current load over a $T_{schedule}$ interval falls below *DeAllocLoad*/2.

When a processor is added to the usable set, its status is set to ACTIVE and its *Processor Advance Time (PAT)* value is set to zero, making it a prime candidate for accepting new clusters. Conversely, processors removed from the usable set are marked as being INACTIVE. PAT is the metric used to trigger load migrations and well be discussed in great detailed in the next section.

7.4.2 Load Balancing Policy

The load balancing policy, attempts to distribute clusters across processors to equalize the rate of progress of each processor through simulated time, taking into account the internal (Time Warp) and external workloads assigned to each processor as well as differences in processor speed (processor heterogeneity). The central metric that is used to accomplish this is the *processor advance time*

```
AnalyzeStatistics()
   for all INACTIVE processors, PE[i]
      if( PE[i].Load < PE[i].DeAllocLoad / 2 )
         PE[i].Status = ACTIVE;
         PE[i].PAT = 0;
      end if;
   end for;
   done = FALSE;
   while( !done )
      compute PAT for all ACTIVE processors by
         summing the CAT values for each
         processor's assigned clusters;
      sort processor statistics based on PAT;
      for all ACTIVE processors, PE[i]
         sort PE[i]'s cluster statistics based
            on the cluster's communications
            affinity to PE[high];
      end for;
      if( TryToOffLoad() == FAILED )
         done = TRUE;
      end if;
      if( PE[high].NumClusters == 0 )
         PE[high].DeAllocLoad = PE[high].Load;
         PE[high].Status = INACTIVE;
      end if;
   end while;
end AnalyzeStatistics;
```

Figure 7.1: Background Execution Algorithm.

(PAT). The processor advance time indicates the amount of *wall-clock time* required for a processor to advance one unit of simulated time in the absence of rollback. Since we are only interested in obtaining a measure of "useful" work, the BGE algorithm only considers committed or "useful" computation and rolled back computation is not allowed to be treated as additional computation load. Our reason for excluding "non-useful" work in our metric is because these computations are application dependent and not always caused by workload imbalances (internal or external). In fact, even if the load is balanced, Time Warp systems can still experience a cascade of rollbacks caused by the application [Lubachevsky et al. 1991]. These rollbacks create an unstable situation where rollbacks become geometrically longer and ultimately result in significantly longer execution times than the sequential simulation. Consequently, if these "non-useful" computations are included in the metric they might fool the system into making a wrong load balancing decision.

The load balancing policy moves clusters from processors with large PAT values to those with lower values with the goal of minimizing the maximum difference between the PAT values in any pair of processors. PAT values are easily measured for the current mapping of clusters to processors.

```
**TryToOffLoad()**
  highest = NumActiveProcessors-1;
  **for all** clusters, c, assigned to PE[highest]
    **for all** ACTIVE processors, PE[i], 0 to highest-1
      **if**( moving cluster, c, from PE[highest]
        to PE[i] reduces the difference in
        PAT values between PE[highest] and PE[i]
        && the current PAT value difference >
        $\Theta PAT_{max}$)
        MoveCluster( highest, c, i );
        return( SUCCEEDED );
      **end if**;
    **end for**;
  **end for**;
  return( FAILED );
**end TryToOffLoad**;
```

Figure 7.2: **TryToOffLoad** function. $PE[0..N-1]$ is sorted from lowest to highest PAT value. $PE[NumActiveProcessors - 1]$ is the active processor with the highest PAT value. Processors $PE[NumActiveProcessor..N-1]$ are inactive and have PAT value set of infinity.

However, in order to assess the effect of redistributing clusters, another mechanism is required to estimate how well (or poorly) the load will be balanced if a hypothetical move of cluster(s) between processors is performed. For this purpose, the *cluster advance time (CAT)* metric is defined. CAT is defined as the amount of computation required to advance a cluster one unit of simulated time, again in the absence of rollback.

More precisely, we define:

1. $CAT_{c,i}$ is the estimated amount of computation time required by cluster c to advance one unit of simulation time on host i, measured in seconds.
2. $TWFrac_i$ as the fraction of total CPU cycles that a Time Warp program on processor i was allocated over the last $T_{schedule}$ interval. This measure is used to account for a processor's user-induced, external workload.
3. PAT_i is defined as the sum of all $CAT_{c,i}/TWFrac_i$ values of clusters mapped, or hypothesized to be mapped, to processor i.

Operationally, $CAT_{c,i}$ is calculated as follows:

$$CAT_{c,i} = CPU_{c,i}/\Delta_{GVT} \tag{7.1}$$

where $CPU_{c,i}$ is the amount CPU time used to process *committed* events by cluster c on processor i over the last $T_{schedule}$ interval, and Δ_{GVT} is the change in GVT over the last $T_{schedule}$ interval.

By dividing that result by Δ_{GVT}, we obtain the amount of computation time required to advance cluster c one unit of simulation time. By combining the above definitions, we obtain the following:

$$\begin{aligned} PAT_i &= \sum_{c=0}^{C_i-1} CAT_{c,i}/TWFrac_i && (7.2) \\ &= \sum_{c=0}^{C_i-1} CPU_{c,i}/(TWFrac_i\Delta_{GVT}) && (7.3) \\ &= CPU_i/(TWFrac_i\Delta_{GVT}) && (7.4) \end{aligned}$$

where $CPU_i = \sum_{c=0}^{C_i-1} CPU_{c,i}$.

We derive Equation 7.4 by substituting $CAT_{c,i}$ in Equation 7.1. By performing dimensional analysis on Equation 7.4 we observe that PAT_i does in fact represent the amount of wall-clock time needed to advance processor i one unit of simulation time. We observe that based on its definition, $TWFrac_i = TotalCPU_i/T_{schedule}$ where $TotalCPU_i$ is the amount of total amount of user CPU time given to processor i over the last $T_{schedule}$ interval. Note that $TotalCPU_i$ is equal to CPU_i plus the amount of time spent doing "non-useful" work. Substituting these equations into Equation 7.4 yields:

$$\begin{aligned} PAT_i &= (CPU_iT_{schedule})/(TotalCPU_i\Delta_{GVT}) && (7.5) \\ &= (T_{schedule})/\Delta_{GVT} && (7.6) \end{aligned}$$

which is wall-clock time, represented by $T_{schedule}$ per unit of simulation time, represented by Δ_{GVT}. We allow $TotalCPU_i$ to cancel CPU_i since they both are a measure of CPU cycles consumed.

If cluster c is moved from processor i to processor j, then PAT_i is reduced by the amount $CAT_{c,i}$ and PAT_j is increased by $CAT_{c,j}$. The new PAT values reflect the expected wall-clock time for each processor to advance one unit of simulation time after the move is made.

The load balancing algorithm attempts to minimize the maximum of $(PAT_i - PAT_j)$ over all i and j, shown in Figure 7.1. The algorithm repeatedly attempts to move cluster(s) from the processor containing the largest PAT value. Clusters on the donating processor are scanned in order of highest communication affinity to lowest affinity to the receiving (low PAT) processor (see Figure 7.2). Cluster-processor communication affinity is determined based on message counts over last $T_{schedule}$ interval. This is done to lessen the potential for new remote communications to be introduced into the distributed simulation computation. For each cluster, processors are scanned from low PAT values to high in order to locate a destination for the off-loaded workload. If moving the cluster will result in a reduction in the difference between PAT values, the move is accepted, and the procedure is repeated. If subsequent moves fail to reduce the difference in PAT values, the algorithm terminates and resumes when the next $T_{schedule}$ interval begins.

Because statistics are collected for each processor individually regarding cluster CPU utilization and processor load, no modifications are required of the initial PAT value comparison in order to account for processor heterogeneity. However, we observe that once a cluster c migrates from some processor i to another processor j, we must modify the CPU time consumed by cluster c on processor i to reflect the difference in the relative speeds of processors i and j. To accomplish this we introduce a new parameter, σ_i, which denotes the relative speed of processor i, and is set

```
MoveCluster( srcpe, c, destpe )
    copy PE[srcpe] cluster statistics For c to
        PE[destpe];
    $CPU_{destpe,c} = CPU_{destpe,c}(\sigma_{srcpe}/\sigma_{destpe})$;
    add this move to movelist;
end MoveCluster;
```

Figure 7.3: **MoveCluster** function.

by the user. This parameter is used in the **MoveCluster** function, shown in Figure 7.3. Here, the migrating cluster's CPU time, $CPU_{destpe,c}$ is increased or decreased by the ratio of the source processor's σ to the destination processor's σ. Consequently, if the source processor is twice as fast as the destination processor, then the migrating cluster's CPU time will be doubled. Likewise, if the destination processor is twice as fast as the source processor, then the migrating cluster's CPU time will be reduced by half.

Load migration is only performed if the maximum difference in PAT values between any pair of processors exceeds ΘPAT_{max}, where Θ is a user-defined percentage between zero and one. This avoids performing migrations when the benefit that can be realized by the migration is modest.

7.5 Implementation

Our BGE algorithm is implemented as part of the Georgia Tech Time Warp (GTW) system, which is a parallel discrete-event simulation executive based on Jefferson's Time Warp mechanism [Jefferson 1985]. Currently, it runs on shared-memory machines as well as distributed memory platforms. The initial implementation was developed on a BBN Butterfly, GP-1000 [Fujimoto 1989b]. From there, it has been re-hosted to the KSR, the SGI Power Challenge, and the Sun Solaris multiprocessor platforms. A detailed description of all of GTW's optimizations can be found in [Das et al. 1994].

To enable the shared-memory GTW kernel to execute in a distributed environment and support dynamic load management several significant changes were made. First, a *reflector-thread* is created on each workstation to manage all external communications. Its tasks include the sending and receiving of all GVT, application defined and dynamic load management messages. Application messages or events are marshaled to the Time Warp kernel(s) that are executing on that local workstation via shared-memory. To mitigate any OS overheads, the reflector-thread is user-level and periodically polled by the GTW kernel. PVM [Geist et al. 1993] is used for remote or off-processor communications.

Another change was the piggy-backing of Mattern's GVT algorithm [Mattern 1993] on top of the existing shared-memory GVT algorithm [Fujimoto and Hybinette 1994]. In this arrangement, Mattern's algorithm forces the shared-memory algorithm to be executed on each workstation to determine its local virtual time. This information in-conjunction with a lower bound on all transit messages between consistent cuts is used to approximate GVT.

The last significant change to the shared memory GTW executive was adding support for moving LP-clusters among the different workstations. A well known problem in migrating Time Warp LPs (and thus clusters of LPs) is the fact that each contains a large amount of state. Specifically, each LP maintains a history of state vectors, in case rollback is later required. While phases could be used to address this problem (see [Reiher and Jefferson 1990]), this requires implementing a mechanism for rollbacks to span processor boundaries because a rollback may extend beyond the beginning point of a recently created phase. A simpler, though perhaps more radical, solution is to rollback the entire simulation computation to GVT if any load redistribution is to be performed. This makes migration of Time Warp LPs no more expensive than migrating non-optimistic computations because there is no need to migrate the history information. This approach also has the side effect of "cleaning up" overly optimistic computation. In this sense, this approach is not unlike the mechanism described in [Madisetti et al. 1993] which found such periodic, global rollbacks to be beneficial. Our experiments indicate that this mechanism provides a reasonably simple and efficient mechanism for reducing migration overhead.

We define $T_{schedule}$ as the interval of time used for determining load redistribution decisions, and is a user defined parameter given in seconds. This implementation performs load management synchronously, i.e., a barrier is used to stop all processes once the workload policy program has determined that LP-cluster migrations are necessary. After the distributed simulation is halted, workload policy migrations are performed, and then the simulation is allowed to resume execution.

To calculate $CAT_{c,i}$, $CPU_{c,i}$, as shown in Equation 7.1, must be determined for all clusters in the system. To obtain this value, we employ the use of monotonically increasing, hardware timers[1] and measure the computation time used to process each event. These timers where chosen because microsecond resolution was needed to accurately measure these low granularity computations (i.e., computations that only require 10's of microseconds to execute). Other Unix timers, such as `gettimeofday` and `getrusage` only provide milli-second resolution on platforms such a the SGI, which is insufficient for our needs.

Using the fast hardware timers, $CPU_{c,i}$ is a running sum over the $T_{schedule}$ interval that includes the time to (i) enqueue each event into the pending set of events, (ii) dequeue each event from a cluster's calendar queue, (iii) state saving overheads prior to event processing, and (iv) event processing time. Recall that $CPU_{c,i}$ only includes timing information from committed events during the $T_{schedule}$ interval.

Next, to calculate $TWFrac$, which represents the allocated CPU time as a percentage of elapsed wall-clock time, denoted by $T_{schedule}$, the `getrusage` system call is used. Since a typically $T_{schedule}$ interval ranges between 5 and 50 seconds, the `getrusage` system call provides the required timer resolution in this case. This system call returns information describing the resources utilized by the current process, or all its terminated child processes, including such statistics as CPU time spent in user space, CPU time spent in the operating system, page faults, and swaps. By dividing the user CPU time over the last $T_{schedule}$ interval by $T_{schedule}$, $TWFrac$ is obtained.

The BGE manager (BM) is implemented as a separate stand alone program that currently executes on its own machine. It was designed this way to provide a clean separation between the load management policy and the mechanism needed to support it. Moreover, this design simplifies the implementation by not having to integrate this functionality into the existing GTW executive.

Currently, this version of GTW executes on networks of Silicon Graphics and Sun Solaris uniprocessor workstations and multiprocessor servers.

[1]On the Sun/Solaris machines, the `gethrtime` system call is used and provides microsecond timing resolution. On the SGI/IRIX machines, the hardware timer is memory mapped into GTW's address space and provides half-microsecond resolution.

7.6 Benchmark Applications

For the experiments present in this study, we used the following two benchmark applications.

7.6.1 P-Hold Synthetic Workload

P-Hold is a simulation using a synthetic workload model [Fujimoto 1990d]. The simulation consists of a fixed message population that move among the LPs making up the simulation. The processing of a message consists of computing for a certain amount of time and then sending one new message to another LP with a certain timestamp increment. The distribution of the computation time per event, the timestamp increment, and the LP to which the message will be forwarded are parameters of the synthetic workload.

7.6.2 PCS

A PCS network [Cox. 1990] provides wireless communication services for nomadic users. The service area of a PCS network is populated with a set of geographically distributed transmitters/receivers called *radio ports*. A set of radio channels are assigned to each radio port, and the users in the *coverage area* (or *cell* for the radio port) can send and receive phone calls by using these radio channels. When a user moves from one cell to another during a phone call a *hand-off* is said to occur. In this case the PCS network attempts to allocate a radio channel in the new cell to allow the phone call connection to continue. If all channels in the new cell are busy, then the phone call is forced to terminate. It is important to engineer the system so that the likelihood of force termination is very low (e.g., less than 1%). For a detailed explanation of the PCS model, we refer the reader to [Carothers et al. 1995].

7.7 Internal Workload Experiments

In this section, the results from our experimental study are presented. For all experimental data presented in this section we use 8, 167 MHZ Sun Sparc Ultra-1 workstations running version 2.5 of the Solaris operating system.

7.7.1 P-Hold Configuration

We configure the P-Hold program to have a one of two computation granularities: *null* and *1 millisecond.* In the *null* case, event processing is made as small as possible. It consist of scheduling a single event into the future at a time $t + 1.0$, where t is the timestamp of the event currently being processed. In the *1 millisecond* case, a 1 millisecond delay loop is added to the processing overheads of the *null* event. Like the *null* case, a single event is scheduled into the future at a time $t + 1.0$.

Each LP's initial message population is 25. Each of these events is assigned a timestamp that is exponentially distributed between 0 and 1. The number of LPs is fixed at 2048 making the total message population 51200. These LPs are grouped into 128 clusters, 16 LPs each. These 128 cluster are evenly distributed onto the 8 processors, giving each processor 16 clusters or 256 LPs. The maximum number of clusters a processor can support is 32.

Table 7.1: GTW Performance Comparison using balanced *Null* Event Granularity P-Hold with and without BGE Monitoring.

$ExecTime_{GTW}$	$ExecTime_{GTW-BGE}$	% Difference
1118	1207 ($T_{schedule} = 10$)	+7.96 %
	1188 ($T_{schedule} = 20$)	+6.26 %
	1187 ($T_{schedule} = 30$)	+6.17 %
	1206 ($T_{schedule} = 40$)	+7.87 %
	1184 ($T_{schedule} = 50$)	+5.90 %
	1155 ($T_{schedule} = 60$)	+3.31 %

P-Hold is additionally configured to be self-initiated. When an event is processed where the source LP is different from the destination LP, the destination LP will schedule the next d generations of the event to itself. By d generations, we mean that the child of the event, and the child's child and so on up to d times will be scheduled for the same LP. After d generations of the event have been produced, the destination LP is randomly picked. For the experiments discussed here, d is a number initially generated for each event based on a uniform distribution between 0 and 2000.

For the static workload experiments, a 1 millisecond delay loop was added to the event computation whenever any LP mapped to cluster 0 processes an event. All other LPs process null events. Cluster 0 was mapped to PE 0. This type of workload is an unbalanced *event-granularity* workload.

For the time-varying workload experiments, a 1 millisecond delay loop was added to the event computation whenever any LP mapped to cluster zero processed an event for the first-third of the simulation, then to the second-third of the simulation this delay loop was shifted to LPs in cluster 31 on PE 1. Finally, in the last third of the simulation, the delay loop is shifted to LPs in cluster 63 on PE 3.

7.7.2 PCS Configuration

PCS is arranged with 2048 Cells. Each Cell is configured with 25 to 75 portables per cell, yielding an initial message-population from 51,200 to 153,600 respectively. The number of channels per cell is 10. Call holding times are exponentially distributed with a mean of 3 minutes. Call mobility rate is exponentially distributed with a mean of 1 every 75 minutes. Call inter-arrival times per portable are exponentially distributed with a mean of 10 minutes. LPs are divided into 128 clusters, 16 LPs to a cluster, and 16 clusters to a processor.

In PCS, an irregular *event-population* internal workload is introduced by giving each Cell (LP) mapped to cluster 0, an initial portable-population (i.e., event-population) of 1000.

7.7.3 Rules for Setting Θ and $T_{schedule}$

Recall, Θ determines how sensitive the BGE algorithm is to workload imbalances and $T_{schedule}$ In determining the best values for Θ and $T_{schedule}$, several factors were considered. First, we were concerned that the monitoring of GTW might significantly degrade GTW performance for small values of $T_{schedule}$, which determines how frequently GTW performance data are collected. For each cluster the following information is collected: (i) CPU time, (ii) number of events processed, (iii) number of events rolled back, (iv) number of events events aborted [2], and (v) a cluster communications matrix. The cluster communications matrix for a cluster denotes the destination cluster for every message sent by this cluster over the last $T_{schedule}$ interval.

To quantify this perturbation, we compared the execution times of *null* event granularity P-Hold under balanced internal workload conditions with and without BGE monitoring. Θ was set to 1.0 to eliminate all migrations. The results of this comparison are shown in Table 7.1. We observed that for event small values of $T_{schedule}$ (i.e., 10 and 20 seconds), the perturbation was less than 8%. Consequently, we believe that monitoring overheads are not a significant factor in determining the value for $T_{schedule}$ or Θ.

The next factor to be considered is migration costs, which includes the amount of time required to halt and roll the simulation back to GVT, known as *halt time*, as well as the time to effect the prescribed LP-cluster migrations, called *move time*. We observed move times are affected by the amount of data contained within a cluster as well as how many clusters are moved. Halt times are affected by the degree to which GTW is "out-of-balance" since the more "out-of-balance" GTW is the further some processors must be rolled back during the halt phase.

Moreover, both Θ and $T_{schedule}$ affect total migration costs. As $T_{schedule}$ is increased, the opportunity for migration decision points decrease, which decreases the rate at which clusters could be moved. Likewise, if Θ is increased, the maximum difference in observed PAT values must be greater to effect any cluster migrations, thus reducing the likelihood that any cluster migration will happen, particularly for small to medium sized workload imbalances. As a result, one may have a tendency to set both Θ and $T_{schedule}$ to large values. However, we shall see that other factors such as reaction time and workload sensitivity suggest otherwise.

To better understand how migration costs, reaction time and workload sensitivity interact for particular set of Θ and $T_{schedule}$ values, we conducted experiments where we induced an internal workload on PCS with 75 events per LP and varied Θ and $T_{schedule}$ across a wide range of values to determine the set of values that minimizes execution time. The results of this experiment are shown in Figure 7.4. The execution times are the average over three runs.

We observe that as Θ is increased from 0.05 to 0.15 and $T_{schedule}$ is increased from 5 to 15 seconds, the execution time drops. Please note that the origin of the graph in Figure 7.4 is $\Theta = 0.05$ and $T_{schedule} = 5$ seconds. The reason for this behavior is because when both values are set low, the migration costs overshadow any benefits gained due to an increase in reaction time (small $T_{schedule}$) or sensitivity (small Θ). It is at this point the system is in the "valley" of low execution times and is where the system should operate to achieve the highest possible performance. However, as Θ and $T_{schedule}$ are increased beyond those values, the execution time increases and continues to do so. This behavior is attributed to a slow reaction time, which is caused by increasing $T_{schedule}$, and lower algorithm sensitivity, which is caused by increasing Θ.

Based on our experience with the BGE algorithm, we make the following guidelines for setting Θ and $T_{schedule}$ such that the "valley" of low execution times is discovered in as few runs as possible.

- Initially, configure $\Theta = 0.05$, and $T_{schedule} = 5$. By setting Θ and $T_{schedule}$ low, we avoid having to worry about the BGE algorithm not detecting an unbalanced workload or having a slow reaction time.

[2]As a flow-control mechanism, GTW will "abort" an event if during the processing of that event no memory is available to schedule a future event.

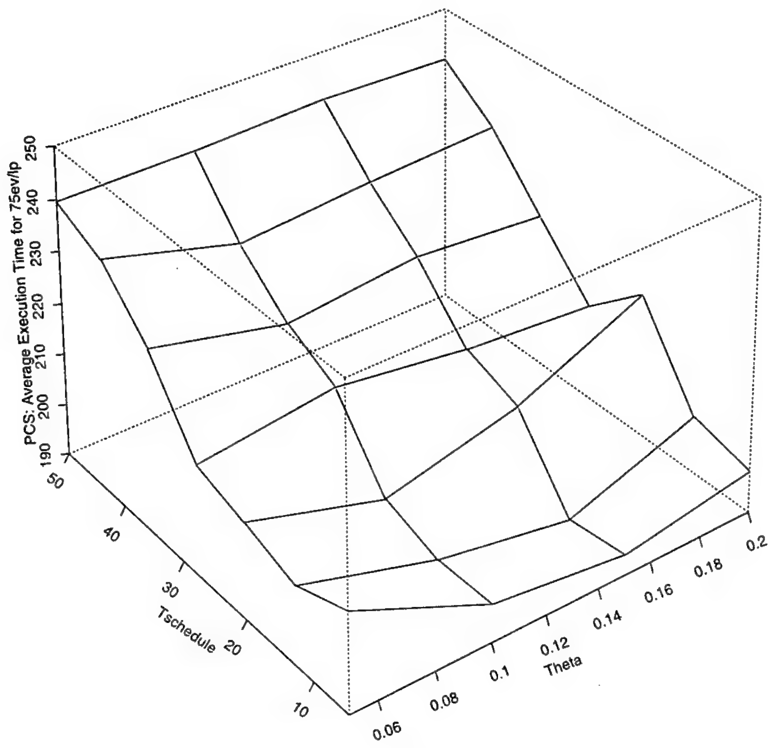


Figure 7.4: *Perspective Plot of PCS, 75 events per LP:* Execution Time as a function of Θ and $T_{schedule}$. The minimum along the Θ axis is 0.05 and for the $T_{schedule}$ axis is 5 seconds.

- If it is observed that the BGE algorithm is consistently initiating cluster migrations every $T_{schedule}$ epoch, particularly when the workload is balanced, increase Θ by 0.05 and $T_{schedule}$ by 5 seconds.
- If the total migration costs are greater than 10% of total execution time, then increase Θ by 0.05 and $T_{schedule}$ by 5 seconds. This will reduce the migration costs without having to be concerned about a slow algorithm reaction time or the BGE algorithm not detecting an unbalanced workload.

7.7.4 Performance Results

In this section we compare the execution time of GTW with and without our BGE algorithm with the introduction of a internal (static and time-varying) workloads. In all cases, the results presented are the average of three runs.

We define *Speedup* as the sequential execution time divided by GTW (parallel) execution time. For all the results presented here, a fast sequential simulator is used, which contains an optimized Calendar Queue [Brown 1988]. The Calendar Queue has a $O(1)$ enqueue and dequeue time for the simulation models tested here. Consequently, our speedup results are very conservative compared to using a $O(log(n))$ data structure, such as the Skew Heap [Sleator and Tarjan 1986] or Splay Tree [Sleator and Tarjan 1985]. Our experiments indicate, that the sequential execution time for *null* event granularity P-Hold is 2.6 times faster when using the Calendar Queue than with the Skew Heap. Similar results where reiterated in a recent, comprehensive study of priority queue data structures by Ronngren and Ayani [Ronngren and Ayani 1997].

Table 7.2: Performance of BGE Algorithm in the Presence of Irregular Internal Workloads For GTW with BGE, $\Theta = 0.15$, and $T_{schedule} = 10$.

Application	Speedup Improvement	Speedup
P-Hold, Static Workload	29%	1.8
P-Hold, Time-Varying Workload	35%	1.8
PCS, Static Workload	127%	2.2

Next, we define *SpeedUp Improvement* as $(Speedup_{GTW-BGE} - Speedup_{GTW})/Speedup_{GTW}$ where $Speedup_{GTW-BGE}$ is the speedup obtained with the BGE algorithm and $Speedup_{GTW}$ is the speedup obtained without the BGE.

As shown in Table 7.2, we observe good improvements in speedup, ranging from about 30% for P-Hold to almost 130% for PCS. However, the overall speedup may seem low. Closer examination of these internal workload reveals that the execution time obtained by GTW with BGE is within 5% of optimal for P-Hold and 50% of optimal for PCS.

To determine the optimal execution time for *P-Hold*, we use a technique called *Critical Path Analysis (CPA)*, developed by Berry and Jefferson [Berry and Jefferson 1985]. CPA provides a means of analyzing the event-dependencies in a parallel simulation, such that the minimum execution time assuming an infinite number processors can be determined. The *critical path* of a simulation is the longest path, measured in real time, through the event dependency graph.

In constructing a dependency graph for the P-Hold model (static workload case) with the 1 millisecond event granularity for all LPs mapped to cluster 0, we note that there are very few interactions between LPs due to the self-initiating nature of this particular model. This allows us to simplify the critical path analysis by assuming that no dependencies exist between LPs (this assumption actually reduces the optimal execution time). Now, because of clustering, all LPs mapped to the same cluster will process events sequentially. Consequently, it is easy to see that the events processed by an LP assigned to cluster 0 form the critical path. This is due to their 1 millisecond event granularity, compared to the 8 microsecond event granularity for events processed in other clusters.

The execution time of the critical path can be calculated as follows. First, there are 16 LPs with 25 initial events each assigned to cluster 0. Each of these events has an initial time stamp of about zero. Upon processing, an event is scheduled for the same LP 1 unit of simulated time into the future. Consequently, to advance 1000 units of simulated time, the LPs of cluster 0 must process $16 * 25 * 1000 = 400{,}000$ events. Each of these events require 1 millisecond of processing time, yielding an optimal execution time of 400 seconds. The execution time reported by GTW with BGE is 418 seconds, which is only 4.5% greater than the optimal.

The critical path in the time varying internal workload, P-Hold model is similar to the static internal workload. Here, the critical path starts with cluster 0 then migrates to cluster 31 and ends with cluster 63. Because the simulation end time was twice as long, the critical path is twice as long, yielding an optimal execution time of 800 seconds. For GTW with BGE, the best execution time is 840 seconds, which only 5% greater than optimal.

Using Amdahl's Law, we observe that the PCS model performs within 50% of the optimal speedup for an infinite number processors with zero overhead for synchronization. Amdahl's Law states that compute time can be divided into the parallel portion and serial portion, and no matter how high the degree of parallelism in the former, the speedup will be asymptotically limited by the serial portion. To use this Law we first find the percentage of the PCS application that must be done sequentially. To do that, we observe that cluster 0 has about 40 times more work to perform per unit of time than the other clusters (computed by 1000 event per LP divided by 25 events per LP). Consequently, cluster 0 comprises $(40/(40 + 127)) = 23.95\%$ of the entire simulation computation, and this computation must be performed serially, since all LPs in this cluster are mapped to the same processor. We also note that cluster 0 is the critical path, if we assume no dependencies. Using Amdahl's Law, the absolute best speedup is then $1/0.23.95 = 4.175$. Given that we are operating in a NOW environment with high communications overheads (i.e. no special purpose communications hardware is employed), a speedup of 2.2 does appear in line with what can be expected.

7.8 External Workload Experiments

In this section, the results from our experimental study are presented. Here, we use 8, 167-MHZ Sun Sparc Ultra-1 workstations running version 2.5 of the Solaris operating system. A workload manager program ensures the external workloads are consistently induced. All results presented are the average over three trials.

7.8.1 P-Hold

Both the *null* and *1ms* event granularity P-Hold models are configured with 2048 LPs and 25 initial messages per LP yielding a total message population of 51200. These LPs are grouped into 128 clusters, 16 LPs each. These 128 cluster are evenly distributed onto the 8 processors, giving each processor 16 cluster or 256 LPs. The above P-Hold configuration parameters are used in all experiments presented here.

7.8.2 PCS

We configure the *call initiated* PCS model with 2048 cells and 25 portables per cell. The number of channels per cell is 10. Call holding times are exponentially distributed with a mean of 3 minutes. Call mobility rate is exponentially distributed with a mean of 1 every 75 minutes. Call inter-arrival times per portable are exponentially distributed with a mean of 10 minutes. LPs are divided into 128 clusters, 16 LPs to a cluster, and 16 clusters to a processor.

7.8.3 Performance Results

In this section, we compare the execution time of GTW with and without our BGE algorithm in the presence of external (static and time-varying) workloads. In all cases, the results presented are the average of three runs. A summary of the performance results is shown in Table 7.3.

For the static workload experiments, we induce three, increasingly larger, static workloads on PE 1 and run each application (*null* P-Hold, PCS, and *1ms* P-Hold). In the first case, a single external task is induced on PE 1. We then induce two tasks on PE 1, and in the third case, four external task are induced on PE 1. Each external task is CPU bound and performs no external I/O.

Table 7.3: Performance of BGE Algorithm in the Presence of External Workloads.

Application	Speedup Improvement	BGE Speedup (4-task case)
Null P-Hold, Static Workload	30% (1-task) to 170% (4-task)	1.8
PCS, Static Workload	40% (1-task) to 180% (4-task)	2.3
1ms P-Hold, Static Workload	50% (1-task) to 260% (4-task)	6.3
Null P-Hold, Time-Varying Workload	28% (4-task, 250 sec.)	1.7
PCS, Time-Varying Workload	20% (4-task, 250 sec.)	2.1
1ms P-Hold, Time-Varying Workload	24% (4-task, 250 sec.)	5.7

For the time-varying workload experiments, we induce two, increasingly larger static workloads on PE 1, with a varying *on* and *off-period* and run each application. In the first case, a 2-task external workload is induced on PE 1. We then induce 4-tasks on PE 1. In each case, we vary the *on* and *off-periods*, but keep them equal. That is to say that the *on-period* is equal to the *off-period.* Consequently, all of the time-varying workloads have a 50% duty cycle. Here, we vary the *on/off-period* among following set of values: 25, 50, 150, and 250 seconds.

First, we present the comparison results for *null* event granularity P-Hold. We observe that GTW with BGE is consistently faster than GTW without it, ranging from 30% faster in the 1-task case to 170% faster in the 4-task case. It was determined that relatively small values of $\Theta =$ and $T_{schedule}$ yield the fastest execution times.

Despite the significant improvements in speedup, GTW with BGE is still only able to obtain a speedup of about 2 on 8 processors. However, given the low event granularity of the P-Hold application combined with the high overheads for sending and receiving messages, these results appear in line with what can be expected.

Next, we compare the performance of GTW with and without the BGE algorithm using the PCS model. We observed a similar pattern to that previously shown for the *null* event granularity P-Hold model, where GTW with BGE for all static workloads yields shorter execution times. However, for PCS, the fastest execution results when $T_{schedule} = 30$ as opposed to $T_{schedule} = 20$ for *null* event granularity P-Hold. The reason this increase in $T_{schedule}$ is because the move costs for PCS are higher because of a larger message size. For PCS, the message data contained in an event is 72 bytes, compared to only 8 bytes for P-Hold.

In this last series of static external workload experiments, we compare GTW with and without the BGE algorithm running *1ms* event granularity P-Hold. Similar to the previous two series of experiments, we again observe that GTW with the BGE algorithm completes *1ms* event granularity P-Hold with significantly shorter execution times than plain GTW, yielding a peak performance improvement of 260% in the 4-task case.

Now, unlike the previous applications, when running 1ms P-Hold, PE 1 was deallocated from the usable set of processors, as shown in Figure 7.5. The primary difference between these applications is event granularity. Consequently, these findings suggest that event granularity plays an important role in determining when a processor should be deemed unusable.

Last, we observe some anomalous cluster allocations occurring during the deallocation of PE 1. It appears another processor during the same epoch has 11 of its 16 clusters redistributed.

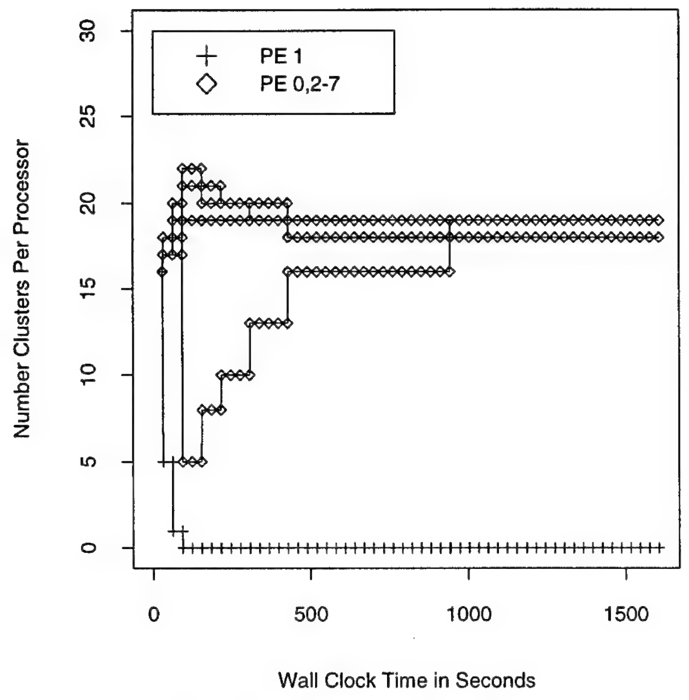


Figure 7.5: *1ms Event Granularity P-Hold:* Cluster Allocation over Time with 4-Static, External Tasks on PE 1.

This behavior is attributed to inaccuracies in cluster CPU utilization times. We will re-visit this phenomenon in the next section and provide a detailed explanation for its occurrence.

For the time-varying workloads, it was determined that $\Theta = 0.15$ and a $T_{schedule} = 50$ did the best overall at consistently detecting and migrating the load imbalance at the appropriate points. However, these settings appear to conflict with the best settings for internal and static external workloads, particularly for $T_{schedule}$. For these workloads, the fastest execution time results when $T_{schedule}$ was set to a much lower value. We attribute this phenomenon to an increase in inaccurate cluster CPU utilizations being reported. These inaccuracies will be quantified later in this section.

For *Null* P-Hold in the presence of the 2-task, time-varying workload, we observe that in each case, GTW with BGE is faster than GTW without BGE, ranging from 10% in the 150-second case to almost 30% faster in the 250-second case. For the 4-task workload, we observed similar speedup improvements.

Now, the observed improvement in speedup may seem low (only about 30% in the 4-task, 250-second case), however, when the amount of lost computation power due to the time-varying workload is considered, this improvement is in line with what can be expected. For these experiments, the 4-task time-varying workload has a 50% duty cycle, thus giving GTW a full 8 PEs half the time and 7.2 PEs when the workload is active. 7.2 PEs is calculated by the fact that GTW on PE 1 is given 20% of PE 1's available CPU cycles (i.e., 1/5 since there are 4 external tasks plus GTW all vying for PE 1), plus the other 7 machines. Thus, on average, GTW is given $(7.2 + 8)/2 = 7.6$ processors, which means that GTW is loosing 0.4 processors, or only about 5% of the available computing power.

Next, we present the time-varying results using PCS. Here, for both the 2-and 4-task, time-varying workloads, the observed performance is somewhat different from that of *null* event granularity P-Hold. The primary difference is in the 50-and 150-second cases. In both cases, we observe

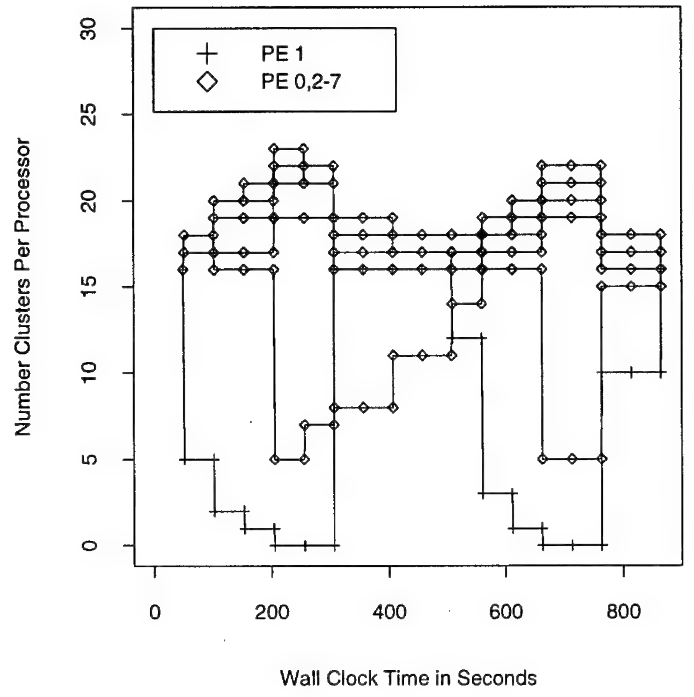


Figure 7.6: *1ms Event Granularity P-Hold:* Cluster Allocation over Time with 4-Time-Varying (250 seconds case), External Tasks on PE 1. For GTW with BGE, $\Theta = 0.15$, and $T_{schedule} = 50$.

little or no reduction in execution time. We attribute this phenomenon to a combination of thrashing migrations every $T_{schedule}$ epoch and higher migration overheads for the PCS. Because of these higher migration costs in the PCS model, only a 20% increase in speedup is obtained for GTW with the BGE algorithm, shown in Table 7.3.

Last, we present the results using *1ms* event granularity P-Hold. Here we observe that GTW with BGE consistently yields shorter execution times than GTW without BGE both the 2-task and 4-task, time-varying external workloads. Even the 50-second case with 4-tasks yields lower execution times, which was not the case for *null* event granularity P-Hold. We attribute these findings to the low migration overheads of the P-Hold model, combined with the 1 millisecond event granularity. Despite, the somewhat thrashing cluster migrations made by the BGE algorithm in the 50-second case, these large event granularities appear to mask them, enabling GTW with BGE to process events at slightly faster event rate than GTW without BGE.

We observe a reasonable increase in speedup given the 5% loss of total computing power due to the 4-task, time-varying external workload. This speedup is attributed to the reduction in number of aborted events which is a consequence of the BGE algorithm migrating clusters off PE 1. In fact, PE 1 is *deallocated* from the usable set during the workload's *on-period*, and then reallocated when workload sleeps, as shown in Figure 7.6.

However, what is surprising about these cluster allocations, is that there appears to be an additional processor that has lost its clusters at the same time PE 1 is being deallocated. This phenomena was also observed in the 4-task, static external workload case for *1ms* event granularity P-Hold. It appears as if there exists an additional time-varying workload being induced on one of the other processors. Further examination of the BGE's trace files reveals that additional external workloads did not exist. So if this phenomena is not the result of an additional external workload,

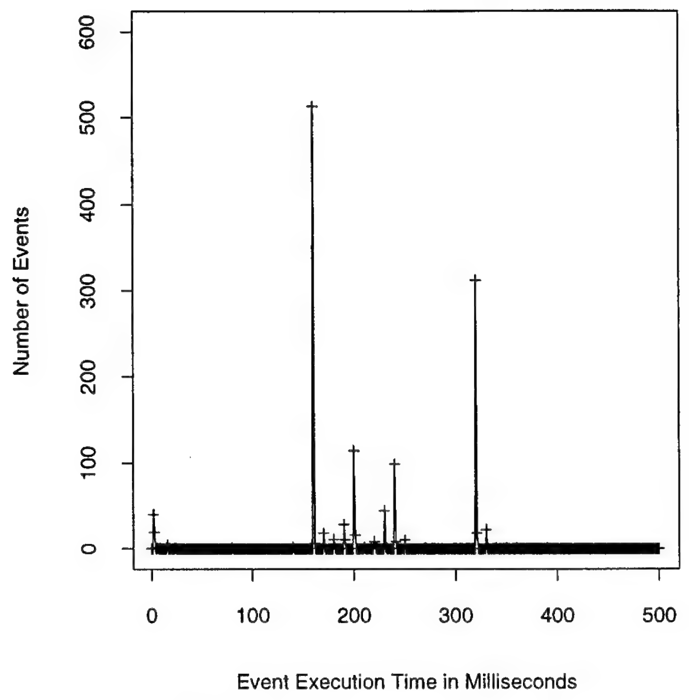


Figure 7.7: *1ms Event Granularity P-Hold:* Histogram of Event Processing Times for loaded PE 1, which as 4-Time-Varying (250 seconds case), External Tasks.

then what is the cause?

Upon closer examination of a the BGE algorithm decisions, the single cluster mapped to PE 1 contains an over-inflated CAT value due to inaccuracies being reported in CPU utilization for that cluster. When this cluster is migrated off PE 1 as a consequence of normal BGE processing, the receiving processor now appears to be over-loaded with work. Consequently, the BGE algorithm begins to migrate clusters off that processor as well.

Now, in many of the previous experiments we have attributed particular anomalous behavior to inaccuracies being reported in cluster CPU utilization. To quantify these inaccuracies, GTW was instrumented to record the number of events and event processing times for the purpose of creating a histogram. We then re-ran *1ms* event granularity P-Hold with 4-task, time-varying external workload on P-Hold. The results from this experiment were very surprising.

The histogram, shown in Figure 7.7, confirms that PE 1 is recording numerous perturbations of event processing times. There were over 500 events recorded with event processing on the order of 160 milliseconds. The actual event processing time should have been only 1 millisecond. In another case, over 300 events were recorded with event processing times in excess of 300 milliseconds. These collective perturbations make PE 1 appear as though it is consuming 75% more CPU time than it is in reality. We attribute these timer perturbations to timed event computations being charged for cycles they do not consume because the hardware clock continues to run while the GTW process has been interrupted by the operating system to allow the other external tasks time to execute.

7.9 Heterogeneous Experiments

The results from our heterogeneous platform study are discussed in this section. For all experimental data presented here, we employed the use of three different kinds of machines: (i) 167 MHZ Sun Sparc Ultra-1 workstation running version 2.5 of the Solaris operating system, (ii) 200 MHZ SGI Indy workstation running version 6.2 of the IRIX operating system, and (iii) 40 MHZ Sun Sparc IPX running version 2.5 of the Solaris operating system.

7.9.1 Determining σ

In order to determine the appropriate value for σ, we compared the sequential execution of the PCS simulation among the various platforms used in these experiments. We determined that the Sun Ultra is about two times faster than the SGI Indy workstation and is about 22 times faster than Sun IPX workstation, thus yielding a σ value of 22.0 for the Ultra workstations, 11.0 for the Indy workstation and 1.0 for the Sun IPX workstation.

7.9.2 Performance Results

In this section, we present the results from the comparison of GTW with and without the BGE algorithm in the presence of heterogeneous processors. Here, we create two different heterogeneous configurations. The first consists of 7 Sun Ultra workstations and 1 SGI Indy. This configuration is referred as *7-fast and 1-slow*. The second configuration consists of 6 Sun Ultra workstations and 1 SGI Indy and 1 Sun IPX. This configuration is dubbed *6-fast and 2-slow*. For these experiments, the PCS model is used.

For both configurations, the optimal value pair set of Θ and $T_{schedule}$ was determined to be 0.15 and 30 seconds respectively. This is the same value set pair that was deemed best for PCS in the static, external workload experiments. These results underscore the observation that processor heterogeneity and static external workloads are in fact duals of each other. That is to say, from the point of view of a Time Warp program, a processor with half the computing power behaves as if it has twice the static external workload as the other processors.

In this first series of heterogeneous experiments, we present the results for the 7-fast and 1-slow configuration. Here, we observe an almost 70% improvement in speedup, which mirror static external workload results.

The cluster allocations for this configuration, shown in Figure 7.8, reveal a picture that is very similar to that found for the static external workloads. Here, we observe the slow processor, denoted by PE 7, having about half its workload removed and redistributed among the other processors. This is what one would expect to happen given that PE 7 is half as fast as the other processors. Consequently, it should be allocated half the number of clusters.

Next, we present the results for the 6-fast and 2-slow configuration. The purpose of this analysis is to see how significant the performance degradation is when we use computing technology that has a large difference in processing capabilities. As previously indicated, in this configuration, the Sun Ultra is about 22 times faster than the Sun IPX workstation. When we compare the performance for the 6 fast and 2 slow configuration with the 7 fast and 1 slow configuration and observed a 25% decrease in speedup. Now, because we are reducing the total amount of computing power by 12%, this accounts for about half the observed performance degradation. The other half is accounted

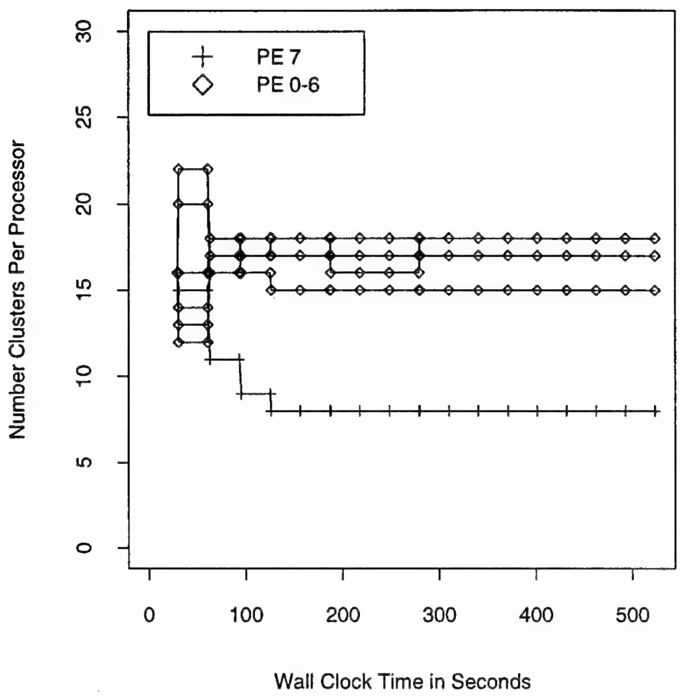


Figure 7.8: *PCS:* Cluster Allocation over Time on 7-Fast Processor and 1-Slow Processor (Best Case). For GTW with BGE, $\Theta = 0.15$, and $T_{schedule} = 30$.

for by the 700% increase in aborted events in the 6 fast and 2 slow configuration. The reason for these aborted events is because Sun IPX with one cluster (see cluster allocations in Figure 7.9, PE 7) is still not progressing fast enough. Recall that the Sun IPX is 22 times slower than the Sun Ultra, however one cluster represents about 1/16 of the workload that is assigned to Sun Ultras. Consequently, the Sun IPX is still over-loaded with work, despite having only one cluster, but is not sufficiently over-loaded to warrant deallocation by the BGE algorithm. Because the Sun IPX is over-loaded, the rate of GVT's advance is slowed, which causes the Sun Ultras to become overly optimistic and abort events. This suggest LP clusters may need to be divided into smaller units of computation, to allow the slower IPX processor to share a smaller portion of the workload.

7.10 Combination-Workload Experiments

For all experimental data presented here, the PCS simulation model is used. The configuration remains unchanged from that previously used in other experiments. The one exception is that for LPs mapped to cluster 0, which is in turn is mapped to PE 0, 1000 initial events are assigned. All other LPs are assigned 25 initial events. This was done to recreate the static internal workload used for PCS experiments presented in Section 7.7. The experiments are performed on 8, 167 MHZ Sun Ultra-1 workstations, where a 4-task, time-varying workload is induced on PE 1. The *on-period* for this workload is 150 seconds, however the *off-period* is only 50 seconds. Thus, this workload has a 75% duty cycle.

We observed that GTW with the BGE algorithm improves speedup by 100% over GTW without BGE. As seen in previous experiments, the BGE algorithm migrates the appropriate number of clusters off the over-loaded processors and redistributing them among the under-loaded processors,

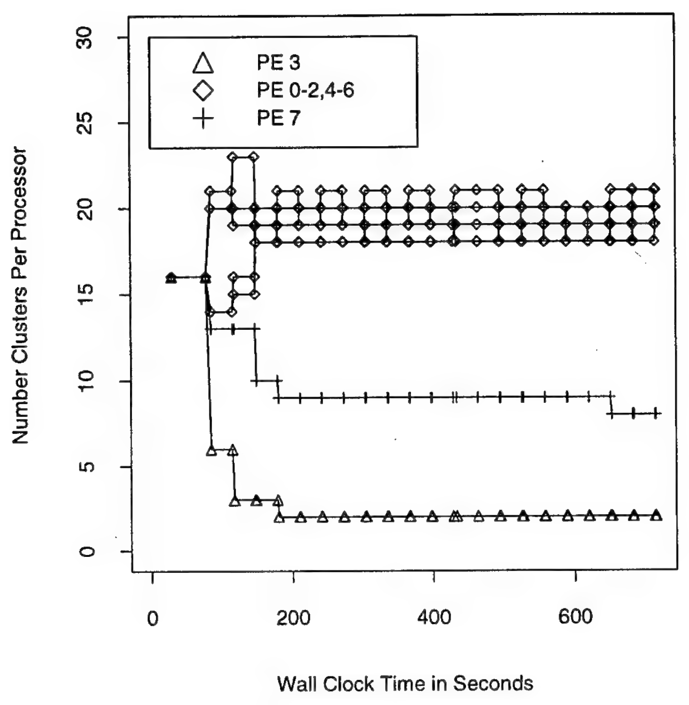


Figure 7.9: *PCS:* Cluster Allocation over Time on 6 Fast Processors and 2 Slow Processor. For GTW with BGE, $\Theta = 0.15$, and $T_{schedule} = 30$.

as shown in Figure 7.10. It is observed that both PE 0 and 1's cluster allocation is reduced to that of a single cluster. However, at 150 seconds into the execution of the simulation, PE 1 is re-loaded with work. This phenomenon is in response to the time-varying external workload on PE 1 going to sleep for 50 seconds.

7.11 Conclusions and Future Work

For Time Warp programs executing on a NOW environment, there are internal and external workload sources that must be taken into consideration if efficient execution is to be maintained. The principal contribution of this work is devising a single algorithm that is able to mitigate both kinds of irregular workloads. The observation driving this algorithm is that in order for a Time Warp program to be balanced, the amount of wall-clock time necessary to advance an LP one unit of simulation time should be about the same for all LPs in the system. In particular, we have demonstrated using a PCS simulation model as well as a synthetic application that our Background Execution Algorithm (BGE) is able to:

- dynamically allocate additional CPUs during the execution of the distributed simulation as they become available and migrate portions of the distributed simulation workload onto these machines,
- dynamically release certain CPUs during the simulation as they become loaded with other, external, computations, and off-load the workload to the remaining CPUs used by the distributed simulation, and

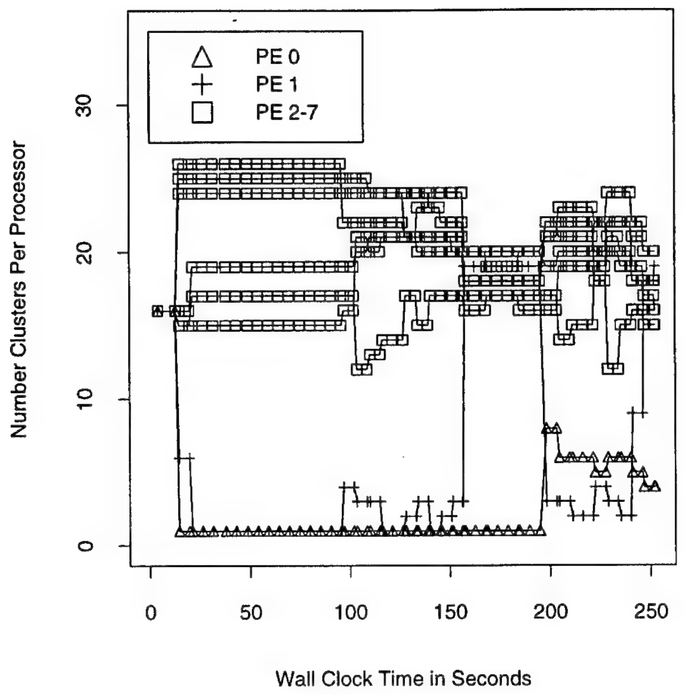


Figure 7.10: *PCS:* Cluster Allocation over Time with Combination of 4-Time-Varying (150/50 second case), External Tasks on PE 1 and Static Internal Workload on PE 0. For GTW with BGE, $\Theta = 0.05$, and $T_{schedule} = 5$.

- dynamically re-distribute the workload on the existing set of processors as some become more heavily or lightly loaded by changing, externally or internally induced workloads.

During the course of this experimental study, two implementation-specific limitations were discovered. The first concerns the inability of our algorithm to detect internal workload imbalances caused by a sharp change in the simulations communication pattern. This limitation is a result of current commercial Unix operating systems, such as Sun Solaris and SGI IRIX, not providing applications a mechanism for determining how much time the operating system spends processing socket operations on the behalf of the application. One solution to this problem is to use a high-performance message-passing system, such as Myrinet [Boden et al. 1995], or Rosu et al's fast ATM firmware [Rosu et al. 1997]. In these systems, the operating system is avoided, placing all message-passing overheads in GTW's address space and thus allowing the communication overheads to be easily obtained by direct measurement. Here, each cluster's CAT value would then be the sum of the amount of wall-clock time spent processing committed events plus the time spent sending and receiving committed off-processor messages.

The second implementation-specific limitation concerns the accuracy of event processing times. Because the timers used are free running, monotonically increasing, hardware clocks, timed event computations may include the time for operating system related tasks or time-slices of other external workloads, should the operating system de-schedule the current running process during the timing of an event.

The obvious solution to these limitations is to modify the operating system such that better performance analysis support is provided. However, this solution lacks portability across many platforms. To ensure application portability, techniques need to be developed that make use of the current operating system interfaces and enable the accurate approximation of system resource utilization. One possible solution we plan to investigate in the future is using context switch statics provided by the `getrusage` system call combined with operating system time-slicing statistics to develop an error potential measurement. This error potential will then be subtracted from the cluster CPU utilization statistics to increase the accuracy of the timing measurements, thus making the BGE algorithm more accurate.

While the focus of this work has centered on efficient execution of Time Warp on NOW platforms, we believe these results are applicable in other synchronization protocols. The key observation made by this work is based on a fundamental truth that not only applies to Time Warp programs, but to *any* data-parallel, distributed simulation synchronization protocol. For example, in a conservative synchronization scheme, if the amount of wall-clock time required to advance an LP one unit of simulation time differs among the various processors, then some form a load imbalance exists.

Finally, when viewed from a higher level, Time Warp load management techniques, such as the one presented here, are stability assurance mechanisms (SAMs), which monitor the Time Warp system and make changes in the behavior of the system to maintain efficient execution. Other SAMs include flow-control, and adaptive memory buffer management techniques. An open question is how do these various SAMs interoperate? At the very least these mechanisms should be designed such that they do not interfere with one another or feed back on each other in a manner that degrades performance. However, an even more interesting question than the issue of interoperability is the existence of a unifying SAM that encompass all types of SAMs for Time Warp systems? What about for all distributed simulation protocols in general? To answer these questions, further investigation is required.

Chapter 8

Visualizing Parallel Simulations that Execute in Network Computing Environments

Parallel discrete-event simulation systems (PDES) are used to simulate large-scale applications such as modeling telecommunication networks, transportation grids, and battlefield scenarios. While a large amount of PDES research has focused on employing multiprocessors and multicomputers, the use of networks of workstations interconnected through Ethernet or ATM has evolved into a popular effective platform for PDES. Nonetheless, the development of efficient PDES systems in network computing environments is not without obstacles that severely degrade simulator performance. To better understand how these factors degrade performance as well as develop new algorithms to mitigate them, we investigate the use of graphical visualization to provide insight into performance evaluation and simulator execution. We began with a general-purpose network computing visualization system, PVaniM, and used it to investigate the execution of an advanced version of Time Warp, called Georgia Tech Time Warp (GTW), which executes in network computing environments. Because PDES systems such as GTW are essentially middleware that support their own applications, we soon realized these systems require their own middleware-specific visualization support. To this end we have extended PVaniM into a new system, called PVaniM-GTW by adding middleware-specific views. Our experiences with PVaniM-GTW indicates that these enhancements enable one to better satisfy the needs of PDES middleware than general-purpose visualization systems while also not requiring the development of application specific visualizations by the end user.

8.1 Introduction

To date, much of parallel discrete-event simulation (PDES) research has focused on employing dedicated multiprocessor and multicomputer platforms to speed up simulation computations, such as the Intel Paragon and KSR machines. However, the use of networks of workstations interconnected through WAN/LANs, such as Ethernet and ATM, has evolved into a popular and effective platform for PDES. The advantages of these network computing environments include (i) ready availability, (ii) low cost, and (iii) incremental scalability. Furthermore, network computing environments retain their ability to serve as a general-purpose computing platform and run commercially available software products.

However, the development of efficient parallel discrete-event simulation systems in network computing environments is not without obstacles. Typically, applications execute on workstations in an open network computing environment whereby each workstation as well as the network itself is subject to uncontrollable external loads. Furthermore, workstations have varying configurations in terms of CPU speed, memory, local verses networked disks, etc. These factors often result in load imbalances and dynamic fluctuations in delivered resources which can be a major source of performance degradation [Schmidt and Sunderam 1994].

To better understand how these factors degrade simulator performance and help develop algorithms that mitigate them, we propose the use of *Graphical Visualization (GV)*. GV has been shown to be a useful aid in performing several activities associated with parallel computing such as verification, performance analysis, and program understanding [Kraemer and Stasko 1993; Simmons et al. 1989]. The usefulness of GV for these activities stems from the highly developed image processing system possessed by humans, which allows us to track multiple complex visual patterns and to easily spot anomalies in these patterns. Consequently, the textual equivalent of the information provided by a visualization may be much more difficult for a user to assimilate. GV systems operate in one of three modes: (i) *on-line*, (ii) *off-line*, and (iii) a combination of (i) and (ii). In on-line mode, the GV system collects run-time performance data from the monitored system and immediately displays the information. In off-line mode, the GV system collects and stores (in a file) the PDES data. After the simulation completes, the stored data can then be displayed or animated. On-line mode has the advantage of showing the "real-time" execution of the monitored system, while off-line mode allows post-processing computations to be run on the collected data that provide a more detailed analysis of the monitored system.

Of great concern was that parallel discrete-event simulators by their very nature are long-running, complex, communication/computation sensitive systems. Consequently, we believe that any PDES visualization system must

- support "on-line" as well as "off-line" modes of operation.
- minimize the amount of computation perturbation such that the GV system does not become a performance bottleneck or mask other performance bottlenecks.
- be easy to use and integrate into the existing system, such that new errors are not easily introduced into the PDES simulator.
- support PDES specific views.
- be robust to endure long-running PDES simulations.

In this case study, we modify an existing general-purpose GV system, called *PVaniM* and used it to conduct a series of visualization experiments on an existing Time Warp system, called *Georgia Tech Time Warp (GTW)* in a network computing environment. PVaniM was chosen because it supported more of the above PDES visualization system requirements than any other GV system to our knowledge. Of particular importance was its ease of use, low system perturbation, and robustness. However, PVaniM's default graphical views did not provide enough insight into the execution of the GTW system. This was due to the fact that all PDES systems are essentially middleware that support their own applications. We define PDES systems as middleware because they typically reside above the operating system and a communication substrate (e.g., PVM) yet support their own applications. We contend that complex network computing middleware, such as Time Warp systems, requires its own *middleware-specific* visualization support. Middleware-specific views focus on illustrating the operation of PDES middleware. They may be used with all simulation applications and allow the user to see how various applications perform on the PDES middleware. It is worth noting that middleware-specific views are not simply a set of application-specific views. Application-specific visualizations for parallel simulations refers to customized views whose appearance is tied to simulation model data. These types of views are typically provided by visual interactive simulation systems, such as those described in Section 8.4 of this paper.

Our experiments demonstrate that by enhancing a general-purpose network computing visualization system with middleware-specific views, one is able to better satisfy the requirements and needs of PDES middleware. Specifically, we document how we have used a GV system to locate performance aberrations, and gained greater insight into how to optimize an existing Time Warp system for a heterogeneous network computing environment.

The remainder of this paper is organized as follows. Section 8.2 describes PVaniM's visualization model, as well as its implementation. Section 8.3 details our distributed GTW system. Section 8.4 discusses related visualization systems. Section 8.5 presents the enhancements made to PVaniM necessary to visualize the GTW system. This newly enhanced visualization system is called *PVaniM-GTW*. Section 8.6 then describes the the results from our effectiveness study and in Section 8.7 presents our conclusions as well as directions for future work.

8.2 Overview of PVaniM

PVaniM is an experimental visualization environment developed for the PVM network computing system [Topol et al. 1997; Topol et al.]. The purpose of PVaniM is to modify and enhance traditional visualization techniques used in multiprocessor and multicomputer visualization environments in order to enable their use in network computing environments.

In performing this adaptation, it became readily apparent that given the nature of clusters, visualization activities can and should be divided into two categories: (i) those tasks that are typically composed of large-grained events and are influenced by and relate to the (dynamic) environment, and (ii) those tasks composed of detailed collections of fine-grained events describing the execution path of the parallel program in question[Topol et al.]. This categorization has resulted in the PVaniM environment following a two-phase approach, with run time or *on-line* monitoring focusing on types of visualization that are mandatory for use during execution, and postmortem monitoring or *profiling* being relegated to detailed program analysis and tuning.

8.2.1 PVaniM's On-line Visualization Support

PVaniM's on-line visualization support consists of the following types of graphical views:

- *Environment Views*–These views provide information regarding the network computing environment in which the parallel application executes. Information such as the workstations used by the parallel application, the external loads present on these workstations, and the amount of memory utilized by the individual tasks are displayed.

- *Performance Evaluation Views*–These graphical displays provide insight into the performance of an application and help locate bottlenecks in the application.

- *Debugging Views*–These views provide insight into the activities of the application (e.g., message passing behavior between tasks) to help locate program bugs.
- *Interaction Views*–These views provide support for outputting results and for interacting (i.e., providing data input) with the parallel application.

8.2.2 PVaniM's Postmortem Visualization Support

PVaniM also provides a substantial amount of postmortem visualization support. PVaniM uses postmortem visualization to provide graphical views for fine grain program analysis, understanding, and tuning. For postmortem visualization, PVaniM relies upon event tracing to provide these displays [Topol et al. 1997]. This can be problematic because the collation of tracefiles from individual tasks can consume network bandwidth and therefore impede the performance of the parallel application. PVaniM has adopted a form of buffered tracing to reduce overheads and increase the efficiency of this activity. Details of PVaniM's implementation may be found in [Topol et al. 1997]. The tracefile is then used to drive PVaniM's detailed, fine-grain profiling views.

As discussed in [Topol et al. 1997], PVaniM's buffered tracing supports a default library of program visualizations such as the one described above, plus it allows the user to develop application-specific visualizations for special needs. Furthermore, PVaniM provides a converter which translates PVaniM trace files to the PICL[Geist et al. 1990] trace format utilized by ParaGraph. This allows PVaniM to also support ParaGraph detailed postmortem views as well.

We are very much aware of benefits offered by postmortem visualization, however they are beyond the scope of this paper. In the remaining sections we focus on how the introduction of on-line middleware-specific views aided in our understanding the behavior of a PDES system in a cluster computing environment.

8.3 Georgia Tech Time Warp (GTW)

Time Warp is an optimistic synchronization mechanism develop by Jefferson [Jefferson 1985] used in the parallelization of discrete-event simulation. The distributed simulator consists of a collection of *logical processes* or LPs, each modeling a distinct component of the system being modeled, e.g., a server in a queuing network. LPs communicate by exchanging timestamped event messages, e.g., denoting the arrival of a new job at that server.

The Time Warp mechanism uses a detection-and-recovery protocol to synchronize the computation. Any time an LP determines that it has processed events out of timestamp order, it "rolls back" those events, and re-executes them. For a detailed discussion of Time Warp as well as other parallel simulation protocols we refer the reader to [Fujimoto 1990a].

As discussed in Section 8.1, cluster computing environments have the following obstacles when used for PDES: (i) high communications latencies that have a detrimental effect on performance of PDES applications and (ii) the reservation of long-term, dedicated computing time is difficult or impossible because of the large user community. To mitigate these obstacles an enhanced version of GTW was built. In this version we added a *reflector thread* that allows GTW to operate in a cluster computing environment, as well as, a *background execution algorithm (BGE)* that detects the presence of external workloads and migrates the appropriate GTW computations such that as fast as possible execution can be maintained in this multi-user, non-dedicated environment.

8.3.1 Reflector Thread

GTW assumes a shared memory model of computation and is composed of a collection of GTW *kernels*. Each kernel is assigned to a host machine and is responsible for processing the events scheduled for the logical processes (LPs) assigned to that host machine. LPs communicate by exchanging timestamped events. Because of the availability of shared memory, the scheduling of an event between two LPs residing on different host machines is done by passing a pointer to the memory location containing the event between the two GTW kernels.

In cluster computing environments the availability of shared memory is far from ubiquitous and requires a significant amount of specialized software and hardware. Moreover, since of the introduction of highly-portable message-passing systems such as PVM [Sunderam 1990] and MPI [Dongarra et al. 1996], message passing seems to be the most prevalent paradigm used to build parallel programs in this environment. Consequently, we were faced with the question of how to unify these different models of computation to enable our GTW system to operate in a cluster computing environment composed of both uniprocessors and shared-memory multiprocessors as well as address the issue of high latency communications.

Our solution to this problem is to create an extra kernel called the *reflector thread* on each host. Its task is to marshal or "reflect" event-messages to and from other GTW systems running on the other host machines. From the view of the other GTW kernels residing on a machine, the reflector thread looks just like any other kernel. When an event-message is scheduled for an LP on another machine, that event-message is given to the reflector thread residing on the sending machine. Once in the reflector thread's hands, the event-message is packaged up and sent to the destination machine using the PVM message passing system where the reflector thread on that machine receives it and places it in the incoming message queue of the GTW kernel that processes events for the destination LP.

In addition to unifying the message passing and shared memory computation models, the reflector thread also hides the latency incurred in sending off machine messages. When on a multiprocessor, the reflector thread executes concurrently with the other GTW kernels, provided there exists a sufficient number of processors. Consequently, when a kernel sends a message it only incurs the small delay of writing into the reflector thread's message queue, and is able to continue processing events without interruption. The reflector thread does all the work in sending and receiving event messages in the cluster computing environment for all GTW kernels on a given host machine.

When on a uniprocessor, the reflector thread must share time with the single GTW kernel. Because of the overheads of context switches and other associated costs, such as an increase in cache invalidations, we are in the process of putting the reflector thread's functionality in the GTW kernel itself.

8.3.2 Background Execution Algorithm

In a network computing environment, not all processors have the same amount of available computing power. Moreover, users will "nice" or execute long running jobs "in background" allowing their CPU intensive jobs to obtain CPU cycles without hogging CPU resources from other users on the system. When the CPU is lightly loaded , the "niced" job will be given more CPU cycles and receive less cycles when the CPU is loaded. Likewise, in a cluster computing environment we anticipate the need for Time Warp programs to share CPU resources among other tasks in a similar fashion. However, a Time Warp program that is well-balanced when executing on dedicated hardware may become grossly unbalanced when executing on machines with external computations. LPs that are mapped to heavily utilized or "slow" host machines will advance very slowly through simulated time relative to others executing on lightly loaded or high-performance host machines. This can cause some LPs to advance too far ahead into the simulation future, resulting in very long or frequent rollbacks. Thus, it is essential that any Time Warp simulation running in "background" take into account the external workloads and host machine processing capabilities.

To mitigate the problems caused by external workloads and heterogenous processing in the network computing environment as well as to allow Time Warp programs to co-exist with other running programs, we developed a background execution (BGE) algorithm that:

- dynamically allocates additional CPUs during the execution of the distributed simulation as they become available and migrates portions of the distributed simulation workload onto these machines,
- dynamically releases certain CPUs during the simulation as they become loaded with other, external, computations, and off-load the workload to the remaining CPUs used by the distributed simulation, and
- dynamically re-distributes the workload on the existing set of host machines as some become more heavily or lightly loaded by changing, internal or externally induced workloads, and at the same time maintain efficient execution of the Time Warp program (i.e limit the amount of rolled back computation).

In keeping with modular design practices, we separate policy from the mechanism used to execute the plan of action. In our GTW system, there is a centralized BGE Policy Program that monitors the factors effecting GTW system performance, such as processor load, events rolled back, and issues LP-cluster migration orders to the GTW kernels. The GTW kernels in turn actually perform the LP-cluster migrations. This policy program may reside on any workstation and typically resides on it own machine to avoid conflicting with any of the other GTW kernels and reflector threads executing on other workstations.

Now, dynamic load distribution of individual LPs burdens simulations containing large numbers (e.g., thousands) of LPs. This is because a large number of entities must be considered by the BGE algorithm, increasing the computation required for load distribution and load balancing information that must be maintained. Further, because migrating each LP requires a certain amount of overhead, independent of the "size" of the LP, migrating many LPs from one host machine to another individually is less efficient than migrating a group of LPs as a unit. Here, LPs are first grouped (by the application) into *clusters* of LPs, and the cluster forms the atomic unit that can be migrated from one host machine to another. In addition to reducing load management and process migration overhead, this approach will group together on the same host machine those LPs that frequently communicate. For a detailed explanation of our BGE algorithm, we refer the reader to [Carothers and Fujimoto 1996].

8.4 Related Visualization Systems

Several simulation systems have utilized visualization in one form or another. Many of these systems are referred to as Visual Interactive Simulation (VIS) systems[O'Keef 1987]. These systems typically provide support for interaction with the simulation model, visualizations of model data, and other model related activities. An excellent example of a VIS system is DISplay[Mascarenhas et al. 1995]. The DISplay system is an application-independent visualization and user-interaction toolkit for use in distributed computing environments. With DISplay, a user may create application-specific displays and interaction facilities that may be used with both parallel simulations or computations.

The WITNESS[Thompson 1993] VIS system provides a visual model builder to reduce the time and complexity of simulation creation. WITNESS also provides graphical reporting elements such as histograms and pie charts that may be displayed while the model executes.

The SIMAN/Cinema[Glavach and Sturrock 1993] system provides a large number of graphical capabilities including barcharts, histograms, and plots. Furthermore, the Cinema system provides CAD support and dynamic status displays and animations may be viewed in real time or in a postmortem fashion.

The XTracker[Bellenot and Duty 1995] system provides graphical visualizations of the execution of parallel simulations. XTracker provides Gantt-like charts of of simulation activities on a per node basis and can also display event messages as traffic between objects.

Finally, [Graham et al. 1996] proposes a visual environment for distributed Time Warp simulations. This system displays information on how individual LPs behave as a means of monitoring simulation performance.

8.5 PVaniM-GTW

PVaniM-GTW, shown in Figure 8.1, is composed of several default PVaniM views and GTW middleware-specific views. These views are updated every t seconds, where t is the user selected *sampling interval*. The sampling interval can be dynamically changed at runtime by sliding the sampling interval interface bar located at the bottom of the PVaniM-GTW window. The sampling interval has a range of one to sixty seconds. We have found that sampling intervals less than 1 second may excessively perturb the monitored program.

The following provides a brief description of both types of views and the insights each contributes to the understanding of GTW's behavioral characteristics and the BGE algorithm. We then describe how the PVaniM-GTW visualization system monitors GTW.

8.5.1 Default Views

To provide a rich selection of information views without overcrowding the PVaniM window space, we display some of the information in *toggled views*. A toggled view can display two different system metrics. Clicking the mouse on the toggled view's title causes the view to change the system metrics being displayed. In the view's title, the performance metric in all capital letters is the one currently being displayed. The list of PVaniM default views includes the following:

- **Host list:** Located in the upper left corner of the PVaniM-GTW window, this view identifies the host machines used by the application and the placement of the tasks, denoted by their

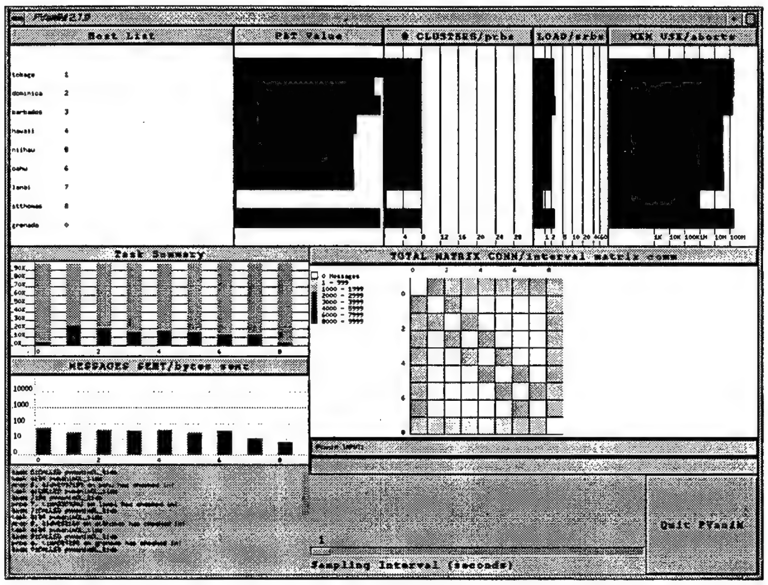


Figure 8.1: PVaniM-GTW graphical user interface. Shows GTW at start-up before the BGE algorithm migrates LP-clusters.

respective integer IDs next to each host name.

Of critical importance to our GTW system, this view allows one to verify a correct machine to task mapping. In fact, this view was instrumental in identifying an unwanted feature that allowed a single uniprocessor workstation to be assigned more than one GTW kernel on successive runs of the GTW system. This "feature" was attributed to PVM's round-robin task assignment scheme. Our fix was to specify an absolute host machine name in the invocation of PVM's `pvm_spawn` routine.

- **Memory usage:** This view is positioned in the upper right hand corner. It illustrates the aggregate amount of memory utilized by the PVM task on each host.

  Time Warp systems are known for using a great deal of memory. Consequently, this view is important in determining if our GTW system is allocating more physical memory than a host machine has to offer, which can lead to extremely poor performance due to thrashing of the virtual memory system.

- **Load:** Located to the left of the **Memory Usage** view, this view provides insight into the aggregate load on the host machines by providing a graphical view of the average number of jobs in the run queue of the host. A novel feature of this view is its ability to account for external loads as well as PVM task loads.

  This view is also important since optimistic simulations, such as our GTW system, can be very sensitive to any reduction or increase in available CPU cycles. This is typically the result of external loads starting or stopping on the host machines used in the cluster-based parallel simulation environment. We have found this view to be instrumental in the understanding of BGE algorithm behavior.

- **Task summary:** Located below the **Host List** view, this view characterizes the percentage of time a task spends performing application-specific computation, denoted in green, the percentage of time a task spends performing PVM sends, denoted in yellow, and the percentage of time a task spends in PVM receives, denoted in red.

  The typical inference drawn from this view is that the program is running well if the task is "in the green", meaning that most of its execution time is being spent doing application specific computation. While this is true for most conservative computations, this view can be misleading for optimistic simulations if used without additional information. The problem is that this view leads one to believe that all computation (shown in green) is good. The fact is that for optimistic simulations, this is simply not true. Because optimistic computations allow out-of-order execution, some computation time will be spent rolling back to put the simulation in a causally correct state.

  The lesson here is that designers and users of visualization systems need to be very cautious and conservative in the inferences they draw from a particular default graphical view, when being used on an optimistic parallel simulation. Moreover, the addition of application or middleware-specific views may be necessary to give a more accurate picture of the system being monitored, as in the case with our GTW system.

- **Messages sent / bytes sent:** Located below the **Task summary** view, this toggled view illustrates how much message traffic a task has incurred over the last statistics sampling interval.

  For our GTW system, this view shows liveness of the simulation program since if no messages are being passed, the GTW system will not progress. Also, this view indicates the size of GTW's event messages, which is an important application characteristic that can effect performance.

- **Task Print View:** Positioned below the **Messages sent** view, this view displays PVaniM and application specific text messages. Using the `pvanimOL_printf` routine, each task can have PVaniM-GTW display their status information. For our experiments here, we used this view to display "raw" data statistics produced by the BGE algorithm as a sanity check against what was being displayed by the middleware specific views, as discussed below.

- **Total matrix / interval matrix communications:** Positioned above the *sampling interval* selection bar, this toggled view gives insight into communications patterns among the active hosts, either cumulative or interval communications patterns.

  In terms of GTW performance characteristics, this view provides insight into how communication patterns change (generally for the worse) when the BGE algorithm is used. This view has lead us to conclude that the BGE algorithm should be modified to consider cluster-cluster affinity information in deciding which cluster to move and where to move them. This phenomena will be covered in more detail in Section 8.6.

8.5.2 Middleware Specific Views

In addition to PVaniM's default set of graphical views, we included the following set of five GTW specific middleware views. To fit the additional views in PVaniM's window space we implemented many of these views as toggled views.

- **Processor Advance Time (PAT) values:** This view is located to the right of the **Host List** view. The *processor advance time* for host machine is defined as the amount of wall clock time needed to advance the simulation a single unit of simulation time. When the PAT values among the the host machines differ, there exists a load imbalance. The BGE algorithm migrates clusters of LPs to the appropriate machines such that the PAT values across all machines should be about equal. Consequently, this view gives an immediate indication how well the BGE algorithm is balancing the load. For host machines not in use, their PAT value is zero.

- **Clusters / Primary Rollbacks (PRBS):** Positioned to the right of the **PAT** view, this is a toggled view that displays either how the clusters are distributed among the active hosts or the percentage of events processed that are rolled back during the sampling interval due to a late arriving or "straggler" application message.

  The clusters view is used in conjunction with other information to determine if the BGE algorithm is operating correctly. Primary rollbacks serve as one of the major indicators for GTW performance. The fewer events rolled back due to straggler messages results in a reduction in erroneous event computations, which ultimately yields better simulator performance.

- **Secondary rollbacks (SRBS):** Located to the right of the **Cluster / PRBS** view, this toggling view (shared with the **Load** view) shows the percentage of the events processed that are rolled back during the sampling interval due to the processing of an *anti-message*. This view provides insight into how far an erroneous computation has spread by indicating the fan-out of LP communication links along which messages are scheduled in the application being simulated.

- **Aborted events:** Right of the **SRBS** view, this toggling view (shared by the **Memory Usage** view) shows the percentage of events processed that are *aborted* during the sampling interval. In the GTW system, a fixed number of event buffers are allocated during initialization and manages those buffers to avoid costly memory allocation system calls during runtime. An event is aborted if the scheduling of a future event fails because all event buffers are currently in use. This approach is used to prevent a host machine from becoming overly optimistic. Usually, events are aborted because a slow GVT calculation process or a general lack of event buffers due to the large set of pending events. Like rollbacks, aborted events have a detrimental effect on system performance and should be avoided whenever possible.

Note, these middleware-specific views do not show LP specific information. It is our belief that the LP-level is to fine of granularity to examine performance particularly for applications where the number of simulations objects is high and event granularity is low. Monitoring these types of applications at the LP level requires the visualization system to keep track of too much data that will ultimately perturb the performance of the Time Warp system. Our approach of aggregating the information on a per host basis reduces data-monitoring requirements while still providing effective visual cues that enable the end-user to draw meaningful and accurate inferences about GTW performance.

8.5.3 Monitoring GTW

To avoid any unnecessary perturbation of GTW, we had to consider not only the level of detail in which the system is monitored, but how should GTW be monitored. The monitoring support for standard PVaniM was designed with an emphasis on ease of use and requiring minimal programmer effort. Essentially, all that is required by the developer is to add an extra header file (GTW is written in C) which provides macros that replace standard PVM routines with calls to PVaniM's monitoring library. The monitoring library performs all activities related to gathering monitoring data and then performs the actions associated with the standard PVM routines.

PVaniM-GTW continues PVaniM's policy of ease of use. All monitoring, including monitoring for middleware-specific visualizations, is performed inside macros that replace standard PVM routines. The routines now perform both generic monitoring support as well as gather middleware-specific monitoring data. The middleware-specific monitoring data is gathered directly out of user space. Registration functions are used to provide the macros with the address of middleware-specific information such as the number of primary and secondary rollbacks, the number of aborts, PAT values, etc.

Several optimizations are utilized to minimize perturbation. First, on-line middleware-specific monitoring data are collected using PVaniM's standard configurable sampling techniques. With these techniques, the user retains control of how often all monitoring statistics are gathered and hence how much perturbation is incurred by the application. Secondly, PVaniM-GTW leverages off of the BGE Policy Program's collection of status updates from each workstation. Since the BGE Policy Program already must gather monitoring information to perform load balancing, PVaniM-GTW gathers its middleware-specific monitoring data directly from the user space of the BGE Policy Program. This avoids the need to recollect information from each workstation that has been already gathered by the BGE Policy Program. This elimination of redundant data-gathering reduces the bandwidth required by the on-line monitoring subsystem.

8.6 Evaluation

The GTW application used in our evaluation experiments is a simulation of a personal communication services (PCS) network. A PCS network [Cox. 1990] provides wireless communication services for nomadic users. The service area of a PCS network is populated with a set of geographically-distributed transmitters/receivers called *radio ports*. A set of radio channels is assigned to each radio port, and the users in the *coverage area* (or *cell* for the radio port) can send and receive phone calls by using these radio channels. When a user moves from one cell to another during a phone call a *hand-off* is said to occur. In this case the PCS network attempts to allocate a radio channel in the new cell to allow the phone call connection to continue. If all channels in the new cell are busy, then the phone call is forced to terminate. It is important to engineer the system so that the likelihood of force termination is very low (e.g., less than 1%). For a detailed explanation of the PCS model, we refer the reader to [Carothers et al. 1995].

Performance Metric	GTW	GTW w/ PVaniM-GTW	Difference
Exec. Time (sec.)	375.89	382.70	+1.81%
Number of Rollbacks	40996	38719	-5.55%
Events Rolled Back	886761	732434	-17.40%
Events Canceled	11611	10113	-12.90%
Events Aborted	4896250	4859281	-0.76%
Avg. Rollback Dist.	21.63	18.92	-12.53%

Table 8.1: Performance results with & without PVaniM-GTW.

The PCS simulation is configured with 4096 Cells giving the simulation a total of 4096 LPs. These 4096 LPs are grouped into 64 clusters of 64 LPs each, and they are mapped to the host machines such that the amount of remote communications is minimized. In all experiments, we used a combination of 4 SGI Indy workstations with 200 MHz MIPS R4400 processors and 4 Sun Ultra1 workstations with 167 MHz Sparc Ultra processors, giving a total of 8 machines used in all experiments. We experimentally determined the Sun Ultra to be 1.8 times faster than the SGI Indy by running a sequential discrete-event simulator and comparing the execution times. The available physical memory on both types of workstations is 64MB. An additional SGI Indy is used to execute the BGE policy program as well as for the PVaniM-GTW monitoring interface.

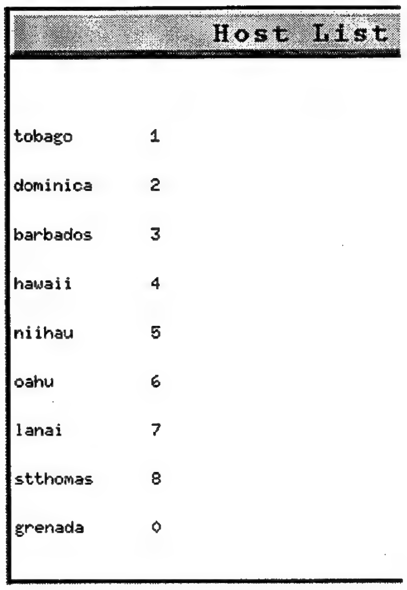
8.6.1 Perturbation Results

As previous discussed, a critical requirement of any online visualization system is that it not perturb the system being monitored. For if it does, any phenomenon observed while using the visualization system could be a side effect of being monitored, thus severally degrading the effectiveness of the visualization system. In this series of experiments we quantify the perturbation caused by the PVaniM-GTW visualization system.

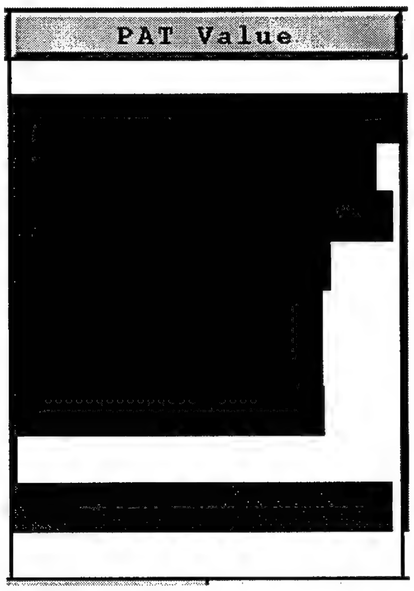
In these perturbation experiments the machines were unloaded by other users and the local area network was lightly loaded. To minimize the potential of masking any of PVaniM-GTW's overheads, we turned the BGE algorithm off. In this mode, the BGE data was collected for use by PVaniM-GTW, but LP clusters were never migrated. To maximize PVaniM-GTW's potential for perturbing the simulation, we set the sampling interval to one second which is currently it's smallest possible value, as well as enabled PVaniM's tracing which is used solely by its postmortem views. Each simulation run processes about 27,000,000 events.

The results of our perturbation experiments are shown in Table 8.6.1. Here, we observed that PVaniM-GTW only increases the execution time by less than 2.0%. Should the sampling interval be increased to 5 or 10 seconds, there would be even less an effect on execution time.

In examining the other performance metrics, aborted events accounted for most of the overhead. Of the 27,000,000 events processed, about 5,000,000 (15%) where aborted. We observe the PVaniM-GTW monitored runs producing about the same number of aborted events as the non-monitored runs (only differs by less than 1%). However, in examining the rollback performance metrics, a much different picture emerges. In the case of events rolled back, the PVaniM-GTW monitored runs are



(a) Host List View



(b) PAT Value View

Figure 8.2: PAT value view *before* migration.

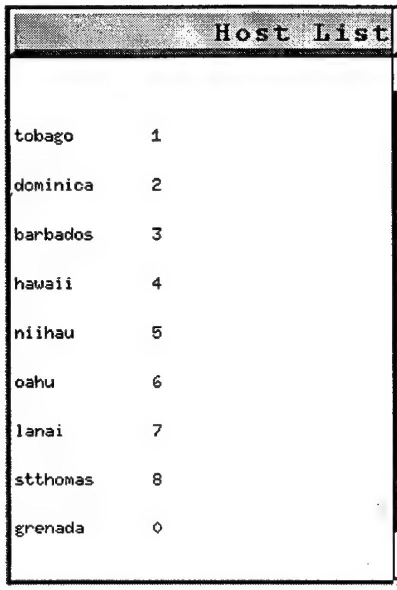
17% lower than the non-monitored runs and the number of canceled events is 13% lower. At first glance is appears PVaniM-GTW is grossly perturbing the rollback performance metrics. However, closer examination of the data reveals that rolled back events account for only 3% of the GTW's overhead computations. Consequently, for these experiments, rollback performance metrics have little effect on overall performance. Thus, in this case, perturbation of these statistics is acceptable.

It is interesting to note that the PVaniM-GTW monitored runs produce less rollbacks and aborted events than the non-monitored runs. The reason for this is because when the monitored GTW system sends an update message it steals CPU cycles that would have been used to process events. This has the effect of increasing the computation granularity, e.g., how much wall clock time it takes to process an event, which slows GTW's rate of progress. By slowing the rate of progress, the likelihood of rollbacks and aborted events is reduced.

8.6.2 Using Combinations of On-line Views

The most significant contribution made by this visualization system is its ability to enable the end-user to draw inferences and make conclusions about the factors effecting the performance of our GTW system and the BGE algorithm. Using PVaniM-GTW , we are able to make the following observations about the BGE algorithm's "hows" and "whys" and provide a behavioral picture of our GTW system and the BGE algorithm. In particular, the following scenario described below demonstrates a case where the GTW system and BGE algorithm do not perform as expected in a heterogenous, network computing environment made up of Sun and SGI workstations. Please note, to make the figures more readable we have "cropped" key views and joined them together. Consequently, the views presented here are not as they would appear in the PVaniM-GTW interface.

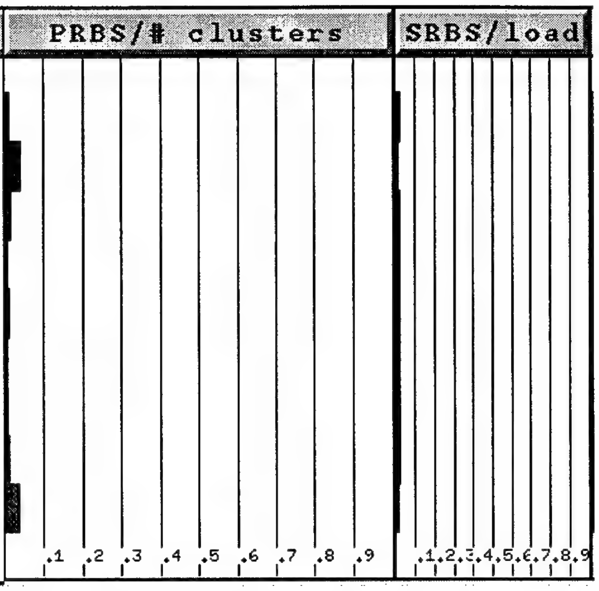
Figure 8.2 shows the PAT values of the GTW system at initial start-up before any LP-cluster



Host List

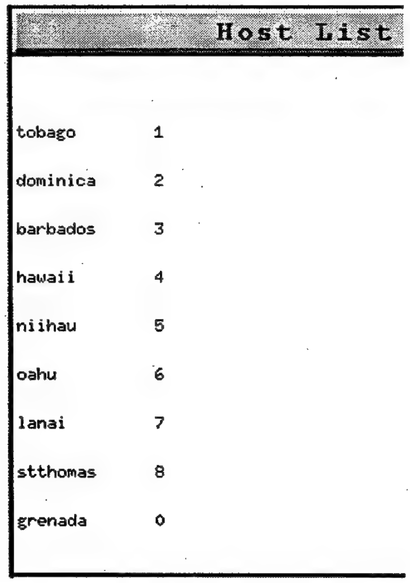
tobago	1
dominica	2
barbados	3
hawaii	4
niihau	5
oahu	6
lanai	7
stthomas	8
grenada	0

(a) Host List View



(b) Rolled Back Events View

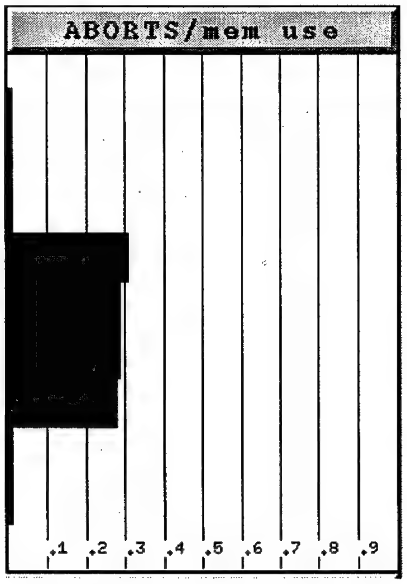
Figure 8.3: Rollback view (primary (PRBS) and secondary (SRBS)) *before* migration.



Host List

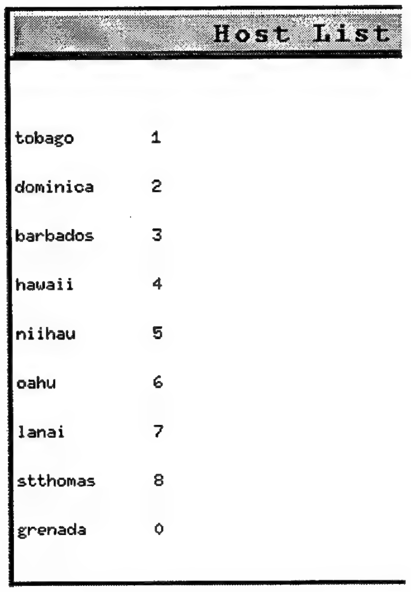
tobago	1
dominica	2
barbados	3
hawaii	4
niihau	5
oahu	6
lanai	7
stthomas	8
grenada	0

(a) Host List View



(b) Aborted Events View

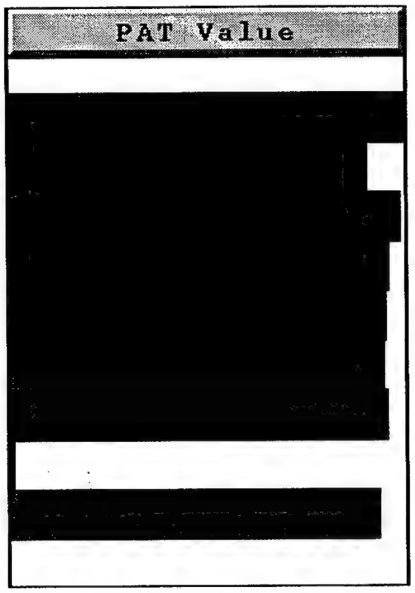
Figure 8.4: Abort view *before* migration.



Host List

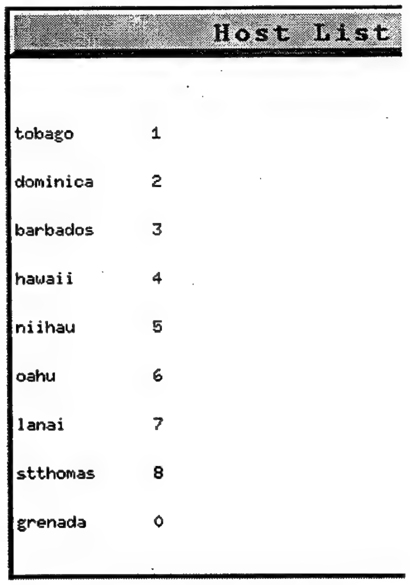
tobago	1
dominica	2
barbados	3
hawaii	4
niihau	5
oahu	6
lanai	7
stthomas	8
grenada	0

(a) Host List View



(b) PAT Value View

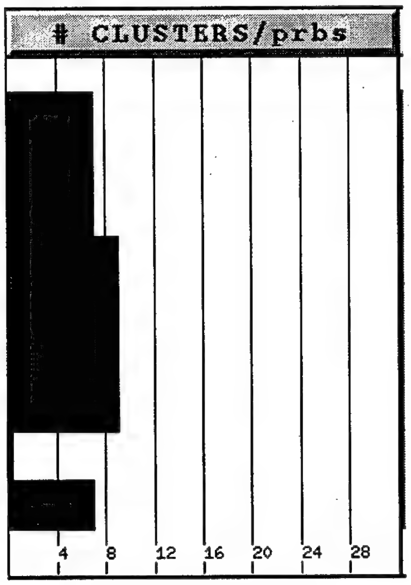
Figure 8.5: PAT value view *after* migration.



Host List

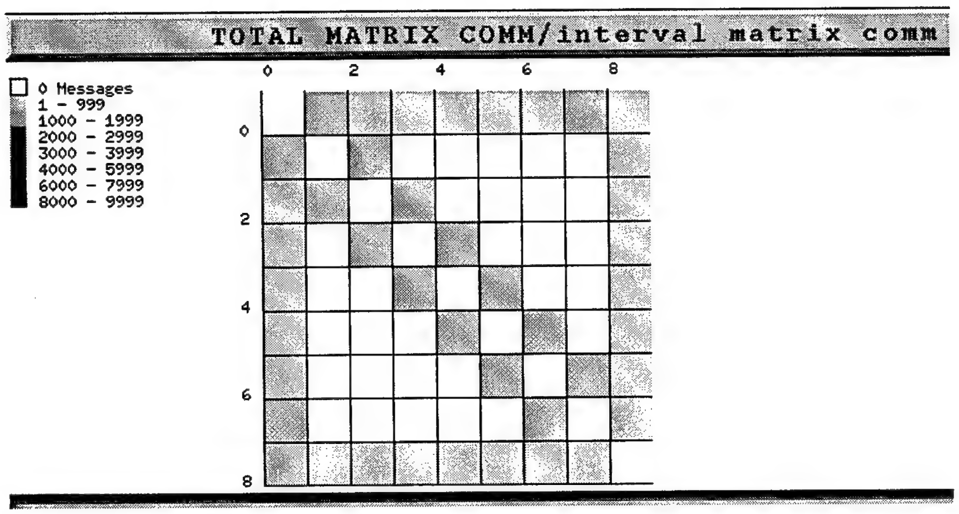
tobago	1
dominica	2
barbados	3
hawaii	4
niihau	5
oahu	6
lanai	7
stthomas	8
grenada	0

(a) Host List View

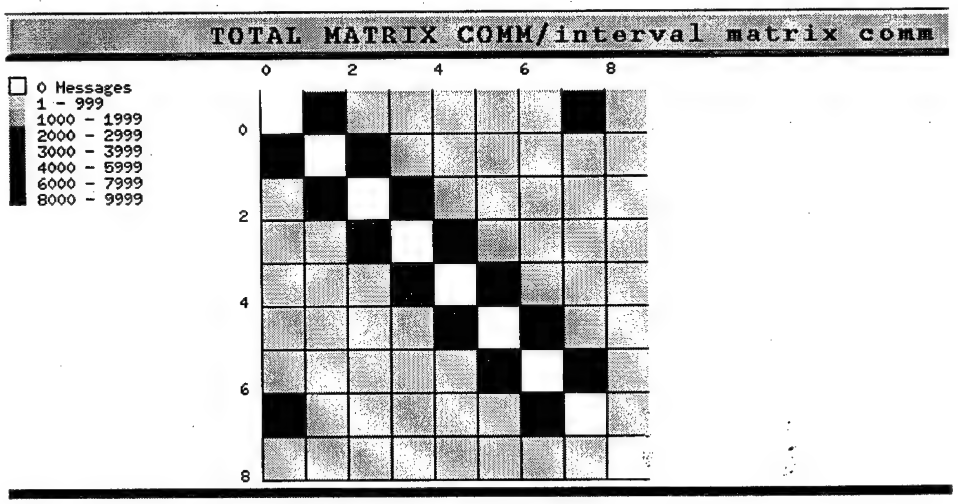


(b) Cluster View

Figure 8.6: Cluster view *after* migration.

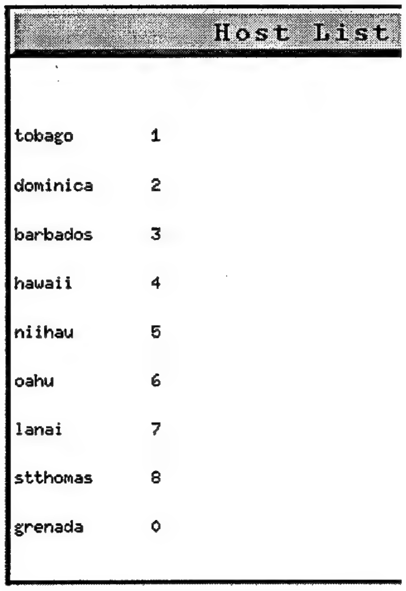


(a) Total Communication Matrix Before Migration



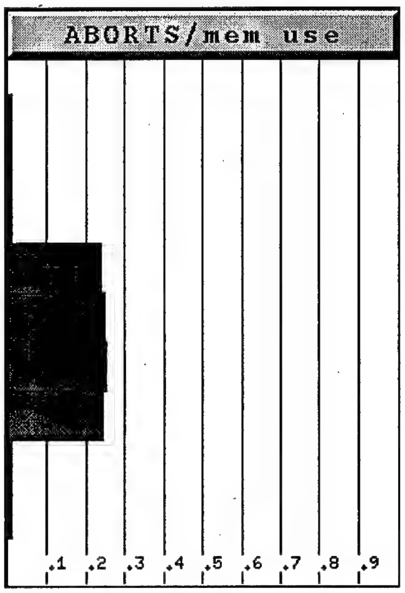
(b) Total Communication Matrix After Migration

Figure 8.7: Total communications views *before* and *after* migration.


Host List

tobago	1
dominica	2
barbados	3
hawaii	4
niihau	5
oahu	6
lanai	7
stthomas	8
grenada	0

(a) Host List View



(b) Aborted Events View

Figure 8.8: Abort view *after* migration.

migrations have been issued by the BGE algorithm. Here, we observe that because of the difference of processing power, the PAT values for host machines $4-7$ (Suns) are much lower than for SGIs [1]. What is surprising, as shown in Figure 8.3, is that the Sun workstations are not rolling back, but instead are aborting a large number of events (30% of all events processed are aborted)(see Figure 8.4). We determined the reason for this behavior lies with GVT being computed at a slow rate, particularly slow for the Suns. Because of this, the Suns are exhausting their supply of memory which prevents them from becoming "overly optimistic" and ensuing a cascade of rollbacks. We are currently examining techniques to speedup the time between successive GVT calculations.

To bring the PAT values back into equilibrium (as shown in Figure 8.5) as well as mitigate the large number of aborted events, the BGE algorithm migrates several clusters off the SGI machines and onto the Suns. Figure 8.6 presents the number of clusters mapped to each node after the migrations are complete. Initially each host machine has 8 LP-clusters mapped to it. After the migrations are complete we observe that the SGI machines donated approximately 25% of their clusters to the Suns. Thus, the SGI now have only 6 LP-clusters, while the Sun workstations now have 10 LP-clusters giving them about 1.7 times the load as the SGI workstations. These migrations are in line with what one would expect given the 1.8 times speed difference between these platforms.

Note, the PAT value views are consistently re-scaled so that the host machine with the highest PAT value is display at the 100% level of the view, such a `tobago` in Figure 8.2(b). All other host machines have their PAT values displayed at the appropriate level relative to the host machine with the highest PAT value. Consequently, it may appear that after a migration round some host machines have an overall higher PAT value than before migration such as the 4 Sun machines in Figures 8.2 and 8.5. However, this is not the case. In these experiments, we not only observed the PAT values being brought in equilibrium, but also a decrease in the highest PAT value after the LP-cluster migrations, which indicates the simulation was progressing at a faster rate than prior

[1]Host 8 (stthomas) is the BGE Policy Program and does not have a PAT value or LP clusters assigned to it.

to performing the LP-cluster migrations. The decrease in overall PAT values was observed in the **Task Print View**, which displays the actual "raw" PAT values as calculated by the BGE Policy Program.

Now, these LP-cluster migrations caused an unexpected side effect. The minimal communication pattern established at start-up (see Figure 8.7(a)) has been corrupted by migration of LP-clusters (see Figure 8.7(b)), resulting in an increase in remote communications. Because of this increase, new computations are introduced into the GTW system which the BGE algorithm had not considered. Consequently, we observe multiple migration rounds before the PAT values are brought into equilibrium. This phenomenon was unexpected since we were already grouping the LPs into clusters with a high degree of communication affinity. We did not anticipate the inter-cluster communications playing such an important role in BGE algorithm performance. We are currently modifying the BGE algorithm to consider inter-cluster communication patterns in the determination of which clusters to move and where to move them.

Moreover, with the PAT values being about equal, we expected to see a sharp drop in the number of aborted events. To our surprise, this expectation did not come true. Shown in Figure 8.8, the number of aborted events for the Sun workstations is still quite high, about 20%, down from 30%. The culprit for this phenomenon was the same as before: the time between successive GVT calculations is still too slow, despite the migration of workload off the slower SGI workstations. We believe by re-integrating the reflector thread's functionality into the GTW kernel for uniprocessor platforms will sufficiently accelerate the rate of GVT calculations to reduce the likelihood of aborted events and ultimately improve GTW's performance.

8.6.3 Sensitivity Tuning of BGE algorithm

Another area in which the PVaniM-GTW visualization system proved instrumental is in the design and selection of the BGE algorithm's sensitivity parameters. In the original BGE algorithm for an LP-cluster migration to be considered effective, it must have reduced the difference in PAT values between the source and destination host machines by an amount greater than the user-defined migration, sensitivity factor, S_m. In experimenting with different settings of S_m, we observed that if the S_m was set too low, the BGE algorithm was overly sensitive to PAT value differences and would always issue LP-cluster migrations. To mitigate this problem we increased S_m. However, in by making the BGE algorithm less sensitive, the PAT values, reported by PVaniM-GTW, were never brought into complete equilibrium and consequently the load across the host machines was not properly balanced. It appeared that there was no happy medium.

However, closer analysis of the BGE algorithm combined with the PAT value view generated by PVaniM-GTW revealed that we were attempting to make two orthogonal decisions using the single S_m sensitivity parameter. The issues in question are as follows:

- Are the PAT values in a global sense sufficiently out of balance such that any cluster migration is necessary?
- Is a given LP-cluster migration effective?

These issues, while seeming similar, must be considered separately. To this end, we modified the BGE algorithm to include a global thresholding parameter, Θ, that examines the global difference in PAT values and determines whether or not GTW's workload is sufficiently out of balance to consider *any* LP-cluster migrations. Operationally speaking, Θ, is a percent difference threshold. Thus, for any LP-migrations to be considered, $(PAT_{max} - PAT_{min})/PAT_{min} > \Theta$, where PAT_{min} is the minimum PAT value and PAT_{max} is the maximum among all host machines. Likewise, S_m is also a percent difference threshold.

To experiment with the setting of Θ and S_m we again employed the use of PVaniM-GTW. With the addition of Θ, we could clearly see from the PAT value view that GTW was not overly sensitive to small load imbalances. Also, when a load imbalances did occur, we visually observed the BGE algorithm responding by migrating the appropriate LP-clusters and the PAT values being brought into equilibrium. Based on our experimental analysis using PVaniM-GTW's view, the proper settings for Θ range between 10% and 15% and for S_m range between 0.5% and 2% for the PCS model. A more comprehensive set of experiments is needed to establish a relationship between the model characteristics and the setting of the BGE algorithm's sensitivity parameters.

8.7 Conclusions and Future Work

While network computing environments are evolving into a popular and effective platform for optimistic parallel discrete-event simulation systems, they are not without difficulties. These environments are subject to uncontrollable external loads and typically composed of workstations of varied computational capabilities. These factors can greatly degrade the performance of PDES systems. This chapter has investigated the use of graphical visualization to provide insight into the execution of a simulator developed for network computing environments. Specifically, our investigation has focused on the augmentation of a general-purpose network computing visualization system with middleware-specific visualizations and has resulted in the PVaniM-GTW visualization system. The synergies provided by the general-purpose graphical views combined with the newly developed middleware-specific views have enabled us to provider more insight into PDES middleware than a generic network computing visualization system. Moreover, the end-user is able to utilize the new insights provided by the system for any simulation application without the expense of developing his or her own custom application-specific visualizations for purposes such as performance analysis and understanding simulator execution. We believe systems such as PVaniM-GTW that employ middleware-specific views provide a cost-effective compromise between both extremes of the visualization tool spectrum. In this case study, PVaniM-GTW has enabled us to identify performance characteristics of the GTW system and weaknesses in the BGE algorithm.

In the future we plan to correct the performance anomalies discovered using PVaniM-GTW and evolve the visualization system to better support clusters of uniprocessor and multiprocessor machines. We also intend to provide collaborative visualization support for GTW simulations.

Chapter 9

RTI Performance on Shared Memory and Message Passing Architectures

This chapter compares the performance of HLA time management algorithms across three computing architectures: (1) a shared memory multiprocessor (SGI Origin), (2) a cluster of workstations interconnected via a low latency, high-speed switched network (Myricomm's Myrinet), and (3) a traditional LAN using TCP/IP. This work is based on the RTI-Kit software package described in a paper presented at the Fall 1998 SIW. This software implements group communication and time management functions on the platforms described above. A time management algorithm designed to efficiently compute LBTS values on shared memory multiprocessors is described. This algorithm exploits sequentially consistent shared memory to achieve very efficient time management. We present results comparing the performance of this time management algorithm with a message-passing algorithm implemented over shared memory.

A goal of this component is to federate Georgia Tech Time Warp (GTW) simulations using RTI-Kit. To facilitate this effort and to exploit trade-offs between performance and network properties we also present results evaluating different communication architectures for networked symmetric multi-processors.

9.1 Introduction

The U.S. Department of Defense (DoD) mandated that the High Level Architecture (HLA) be the standard architecture for DoD modeling and simulation programs [DMSO 1998]. This requires that the HLA span a wide range of applications, computing architectures, and communication paradigms.

Previously most implementations of the High Level Architecture (HLA) Run Time Infrastructures (RTIs) have been implemented on a Network of Workstations (NOWs). Typically these workstations are connected via TCP/IP networks or other high speed interconnects such as Myricomm's Myrinet [The Myrinet 1998].

Recently Shared Memory Multiprocessors (SMPs) have been considered as a platform for HLA RTIs. SMPs offer a different communication paradigm. An HLA RTI implemented over a NOW must use message passing to transfer information between workstations, e.g., using UDP or TCP packets over a TCP/IP network or Fast Messages (FM) [Pakin et al.] over a Myrinet network. In SMPs communication between processors is realized through the use of shared memory. For example, to transfer information from one processor to a second, the first processor writes to a memory location in shared memory and signals the second processor to read from the same location.

In this report a shared memory time management algorithm is presented. This algorithm has been implemented as an RTI-Kit [Fujimoto and Hoare 1998] library and its performance is compared to an existing message passing time management algorithm. In addition we evaluate different communication topologies in order to effectively exploit trade-offs between performance and network properties.

9.2 RTI-Kit

RTI-Kit is a set of libraries designed to support the development of Run Time Infrastructures (RTIs) for parallel and distributed simulation systems. Each library is designed so that it can be used separately or together with other RTI-Kit libraries.

RTI-Kit consists of three main libraries, see Figure 9.1. FM-Lib implements communication

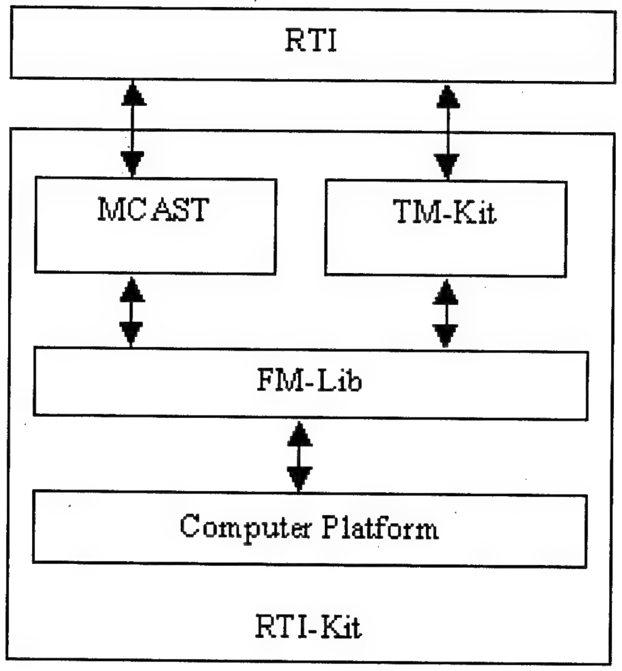


Figure 9.1: Architecture of an RTI using RTI-Kit

over the computer platform. TM-Kit implements time management functions. MCAST implements group communication. Both TM-Kit and MCAST use FM-Lib to carry out their respective functionality. By using a structure such as this it makes it possible to develop RTIs that are independent of the computer platform. To implement an existing RTI on a new computer platform it only becomes necessary to build the appropriate FM-Lib library for that computer platform. To take full advantage of the properties of a particular architecture it is possible for MCAST or TM-Kit to bypass FM-Lib.

The libraries in RTI-Kit are self-contained i.e., the internal structure of one library has no dependence on the internal structure of another. The only dependence between libraries is the interface presented by one library to another. This makes it straightforward to develop a new library with the same functionality as an existing library. As long as the interface between libraries is maintained no changes to other libraries will have to be made.

For the experiments performed for this report several variations of RTI-Kit were developed to operate on different computer platforms. Each of the variations differs only in the version of FM-Lib and TM-Kit used.

9.3 TM-Kit

TM-Kit is a library that implements time management functions. The primary purpose of TM-Kit is to compute the Lowest Bound Time Stamp (LBTS); LBTS is the lowest time stamp of a message that a processor can expect to receive from another processor in the future.

When calculating LBTS a mechanism is needed to determine the global minimum time. In order to determine the global minimum time each processor needs some way to communicate its local minimum time to the other processors. Here we present two algorithms that use different communication paradigms to calculate LBTS. The first algorithm uses message passing to communicate, and the second algorithm uses shared memory.

Each of these algorithms is implemented as a TM-Kit library. The message passing algorithm will be referred to as Message Passing TM-Kit (MP TM-Kit), and the shared memory algorithm will be referred to as Shared Memory TM-Kit (SHM TM-Kit). The architecture of RTI-Kit using MP TM-Kit is exactly the same as that shown in Figure 9.1. However, SHM TM-Kit does not use FM-Lib, instead it is directly linked to the computer platform. In Figure 9.1 this would be seen as a double arrow between the TM-Kit box and Computer Platform box and no double arrow between the TM-Kit box and FM-Lib box.

9.3.1 Message Passing TM-Kit (MP TM-Kit)

When an LBTS computation is started using MP TM-Kit the processors have to send their local minimum time to the other processors by using communication primitives defined by FM-Lib. This algorithm works by having a processor send its local minimum time to another processor at each step of the LBTS computation. When a processor receives a message containing the local minimum time of another processor it computes the pair wise minimum between its local minimum time and the local minimum time contained in the message received. This continues for log N steps, where N is the number of processors. The computation is completed when a processor has received a message containing the local minimum time from log N other processors. The LBTS is the result of the final pair wise minimum operation performed. A more complete description of this algorithm can be found in [Fujimoto and Hoare 1998].

9.3.2 Shared Memory TM-Kit (SHM TM-Kit)

Unlike the previous version of TM-Kit, SHM TM-Kit does not use the FM-Lib library to send messages between processors to calculate LBTS; all communication is done implicitly through the use of variables in shared memory. As stated earlier, SHM TM-Kit communicates directly with the computer platform and not through FM-Lib.

This algorithm assumes a sequentially consistent memory model for memory access [Lamport 1979]. A sequentially consistent memory model is one in which memory accesses by a processor to memory appear to occur in some global ordering and the references by each processor appear to occur in the order specified by that processor. For example, if processor 1 issues memory references M1, M2, M3 and processor 2 issues memory references Ma, Mb, Mc then M1, Ma, M2, Mb, Mc, M3, is a sequentially consistent ordering of the memory references. However, Ma, Mc, M1, M2, Mb, M3 is not because memory reference Mc appears before Mb, and this is not the order specified by the program. This memory model allows the development of simpler more efficient shared memory algorithms than would be possible using a message passing algorithm.

The computation of the LBTS is separated into three stages corresponding to the three critical procedures needed to implement the algorithm. They are 1) starting an LBTS computation, 2) sending messages, 3) reporting local minimum time and calculating LBTS. This algorithm is based on the Global Virtual Time algorithm presented in [Fujimoto and Hybinette 1997].

Required Variables

All of the communication required to compute the LBTS between the processors is done through three variables in shared memory: GVTFlag (an integer), PEMin (an array to store the local minimum time of each processor), and GVT (stores the newly computed LBTS). In addition to the variables in shared memory each processor needs two variables in local memory SendMin and LocalGVTFlag. Their use will be explained shortly.

Stage 1: Starting an LBTS Computation

To start an LBTS computation a processor first tests if GVTFlag is equal to zero. If GVTFlag is equal to zero it indicates that no other LBTS computation is in progress and therefore it is safe to start a new LBTS computation by setting GVTFlag equal to N, where N is the number of processors. The process of testing and setting GVTFlag is a critical section, i.e., only one processor is allowed to test and set GVTFlag at a time to eliminate the possibility that two processors can set GVTFlag equal to N simultaneously.

Stage 2: Sending Messages

In order to calculate the LBTS a processor keeps track of the minimum time stamp of sent messages; this value is stored in SendMin. Though SHM TM-Kit does not use FM to send any messages other libraries of RTI-Kit may use FM so it is vital that SHM TM-Kit be aware of these out going messages to accurately compute the new LBTS. Recording the minimum time stamp of out going messages is only done when an LBTS computation is in progress, GVTFlag is greater than zero, and the processor has not yet reported its local minimum time.

Stage 3: Reporting the Local Minimum Time and calculating the LBTS

Periodically each processor has to give time to TM-Kit to test GVTFlag to determine if an LBTS computation is in progress. The most natural location to place this test is in the main event scheduling loop of the simulation. If the processor does not perform this test often the completion of an LBTS computation will be delayed degrading performance.

Each processor first saves a local copy of GVTFlag by setting LocalGVTFlag equal to GVTFlag. The processor then processes any pending incoming messages. Despite the fact that SHM TM-Kit does not use FM-Lib for communication the other libraries of RTI-Kit may be using FM. It is important to allow the processor to process any messages that have arrived via FM as these messages may affect the local minimum time of the simulation. Then if LocalGVTFlag is greater than zero and if this processor has not yet reported its local minimum time it reports the local minimum time as the minimum of SendMin and the time reported by the RTI. The minimum time reported by the RTI is obtained via a call back into the RTI. The local minimum time is placed in the processor's entry in PEMin.

Next, GVTFlag is decremented and the processor to decrement GVTFlag to zero is responsible for computing the new LBTS. The new LBTS is the minimum of all the entries in PEMin and this value is placed in GVT. This process of reporting the local minimum time and calculating the new LBTS is a critical section so only one processor can perform this operation at any given time. Once the new LBTS value is computed each RTI is notified through a call back into the RTI.

```
Constants
     int N;      /* number of processors */
     int Pid;    /* processor id number  */

Global Variables
     int GVTFlag;
     TM_Time  PEMin[N];  /* local minimum of each  processor */
     TM_Time GVT;        /* computed LBTS */

Local Variables
     TM_Time SendMin;
     int LocalGVTFlag;

Procedure to start an LBTS computation
   StartLBTS()
     begin critical section
          if (GVTFlag =  0) then
               GVTFlag = N;
          end if
     end critical section

Procedure to be called when sending a message. TS is the time stamp of the
message being sent.
   TM_Out(TS)
       if ((GVTFlag > 0) and
          (haven't already computed local min)) then
               SendMin = min(SendMin,TS);
       end if

Procedure to be called in the main event processing loop.  RTIMinTime is
the minimum time reported by the RTI.
   TM_Tick()
       LocalGVTFlag = GVTFlag;
       Process any pending incoming messages
       if ((LocalGVTFlag > 0) and
           (haven't already computed local min)) then
            begin critical section
                 GVTFlag = GVTFlag - 1;
                 PEMin[Pid] = min(SendMin,  RTIMinTime);
                 if (LocalGVTFlag = 0) then
                     GVT = min(PEMin[0] ... PEMin[N]);
                 endif
             end critical section
        endif
```

Figure 9.2: Implementation of LBTS algorithm in SHM TM-Kit

9.4 FM-Lib Library

The FM-Lib library provides primitives to facilitate communication between participants in a simulation. Different versions of FM-Lib have been implemented to operate on different computing platforms and communication networks. An RTI can be made to run on a different platform with no change to the RTI by simply plugging in the appropriate FM-Lib for the platform in use.

Three version of FM-Lib have been implemented Myrinet FM-Lib (MYR FM), TCP FM-Lib (TCP FM), and Shared Memory FM-Lib (SHM FM). All three versions of FM-Lib implement the FM interface.

MYR FM uses FM written for Myrinet. Myrinet is a high speed, low latency switched interconnect network with a bandwidth of 640 Mbits/sec. TCP FM uses a traditional TCP/IP network for communication.

Unlike the previous two versions of FM-Lib SHM FM does not use a network to communicate instead communication is achieved through the use of shared memory. An SMP is a computer with $N(> 1)$ processors and a pool of memory that each processor has access. Each processor that is participating in the simulation allocates memory from the shared memory to act as a message in buffer. When processor a wants to send a message to processor b, processor a writes to processor b's in buffer. When processor b detects a message in its in buffer it reads the message. By using this recipe SHM FM achieves the same behavior inherent to MYR FM and TCP.

9.5 RTI Implementations

Using the two TM-Kit and three FM-Lib libraries four implementation of an RTI were developed to gather performance results. The RTI used, called DRTI, implements a subset of the HLA Interface Specification services. For example, Time Advance Request, Time Advance Grant, Next Event Request, Update Attribute Values, Reflect Attribute Values are implemented in DRTI. Update Attribute Values and Reflect Attribute Values uses attribute handle pair sets to transmit attribute updates.

The four implementations implemented differ only in the FM-Lib and TM-Kit used. Figure 9.3 shows each of the four implementations.

Implementation	FM-Lib	TM-Kit
MYR DRTI	MYR FM	MP TM-Kit
TCP DRTI	TCP FM	MP TM-Kit
MP DRTI	SHM FM	MP TM-Kit
SHM DRTI	SHM FM	SHM TM-Kit

Figure 9.3: Various implementations of DRTI

MYR DRTI was executed on a network of eight Ultra SPARC workstations connected via Myricomm's Myrinet network. Myricomm's Myrinet network is a high speed, low latency switched interconnect network with a bandwidth of 640 Mbits/sec. TCP DRTI was executed on a network of eight Ultra SPARC workstations connected via TCP/IP network. The TCP/IP network is a standard 10BaseT Ethernet with a bandwidth of 100 Mbits/sec. Both MP DRTI and SHM

DRTI were executed on an SGI Origin 2000 Symmetric Multiprocessor [Silicon Graphics, Inc 1998]. The SGI Origin 2000 used is configured with sixteen 195MHz R10000 processors with 4Mbytes of secondary cache and 4Gbytes of shared memory and provides a sequentially consistent model for memory access.

9.6 Performance Measurements

Three benchmarks are used to evaluate the performance of the various implementations of DRTI. The first benchmark established the performance of the underlying communication medium used. The other two benchmarks directly compare the performance of the different implementations of DRTI. The Latency benchmark measures the performance of the communication services of an RTI. The Time Advance Grant benchmark measures the performance of the time management services of an RTI.

9.7 Underlying Communication Performance

The FM-Lib library used makes a significant impact on RTI performance. To better understand the performance results relating to RTI performance a comparison of the three FM-Lib libraries are in order. Figure 9.4 shows the one-way latency of sending a message of size N. The one-way latency is

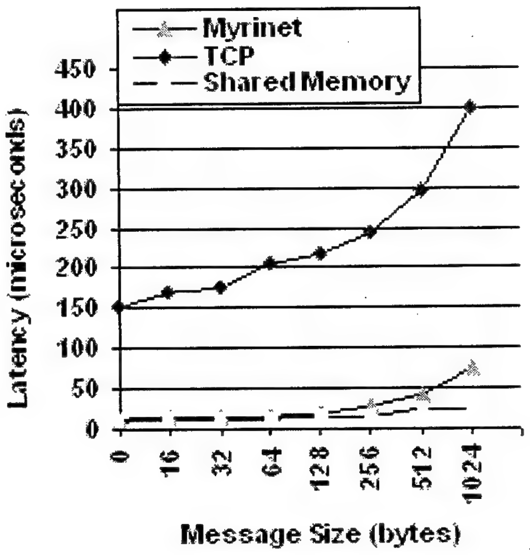


Figure 9.4: Latency measurements for three FM-Lib implementations

obtained by measuring the round-trip latency of sending a message of size N between two nodes and divided by two. SHM FM achieves the lowest latency for sending a message followed by MYR FM and then TCP FM. However, note that the performance of SHM FM is only slightly better than the performance of MYR FM. The performance of DRTI with respect to the Latency Benchmark and Time Advance Request Benchmark will closely follow this relationship. MP DRTI and SHM DRTI will have similar performance results to MYR DRTI and all three show a significant performance increase over TCP DRTI.

9.8 The Latency Benchmark

The latency benchmark measures the latency for best effort, receive ordered communications. This program uses two federates, and round-trip latency was measured. Specifically, latency times report federate-to-federate delay from when the UpdateAttributeValues service is invoked to send a message containing a wallclock time value and a payload of N bytes until the same federate receives an acknowledgement message from the second federate via a ReflectAttributeValues callback. The acknowledgement message has a payload of zero bytes. Latency measurements were taken and averaged over 10,000 round trip transmissions. The one-way latency time is reported as the round trip time divided by two.

Figure 9.5 shows the results for the two implementations running on the network of Ultra SPARC workstations. As expected the performance of MYR DRTI is an order of magnitude better than is

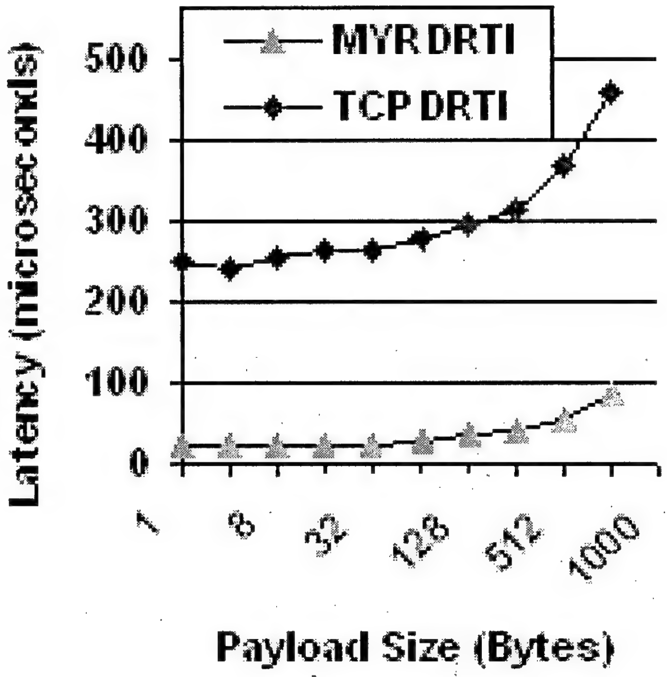


Figure 9.5: Latency measurements for Myrinet DRTI and TCP DRTI

TCP DRTI. Myrinet avoids the cost of having to marshal data through a protocol stack.

The results for the two implementations running on the Origin 2000 is shown in Figure 9.6. The performance of the two implementations is nearly identical. Since the time required to send attribute updates is being measured and both versions are using SHM FM latency times are expected to be the same.

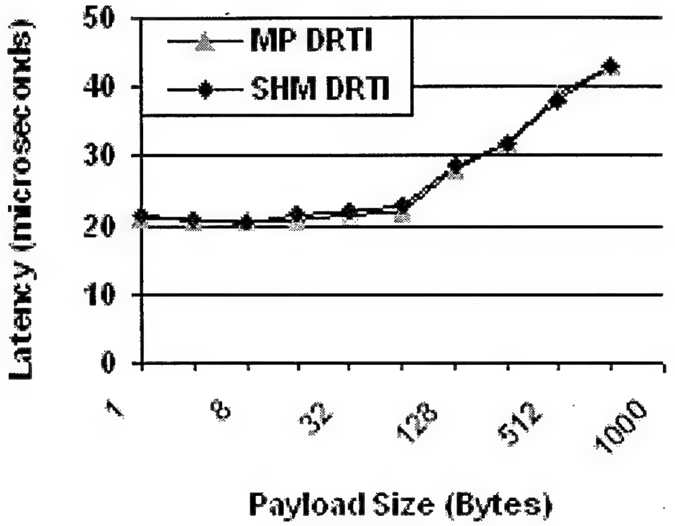


Figure 9.6: Latency measurements for MP DRTI and SHM DRTI

9.8.1 The Time Advance Request Benchmark

The TAR benchmark measures the performance of the time management services, and in particular, the time required to perform LBTS computations. This benchmark contains N federates, each repeatedly performing TimeAdvanceRequest calls with the same time parameter, as would occur in a time stepped execution. The number of time advance grants observed by each federate, per second of wallclock time is measured for up to eight processors.

Figure 9.7 shows the number of TAGs per second achieved by MYR DRTI and TCP DRTI. Again the performance of MYR DRTI is orders of magnitude better than TCP DRTI and as before

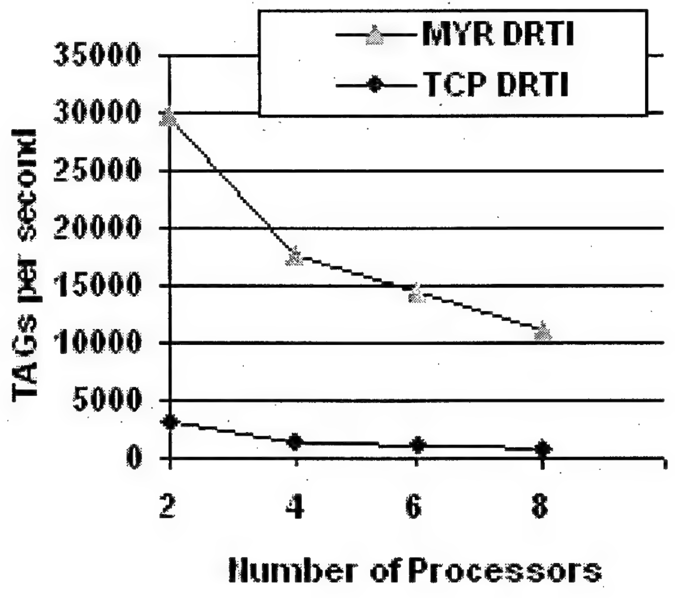


Figure 9.7: Time Advance Request measurements for Myrinet DRTI and TCP DRTI

is due to the Myrinet's low communication overhead.

Figure 9.8 shows the number of TAGS per second achieved by MP DRTI and SHM DRTI. In

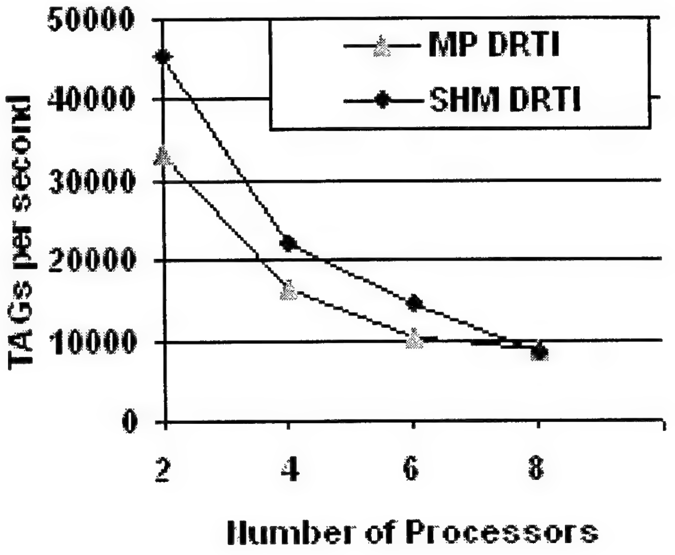


Figure 9.8: Time Advance Grant measurements for MP DRTI and SHM DRTI

both implementations SHM FM is used yet SHM DRTI achieves higher performance. The difference between the two implementations is the time management algorithm, SHM TM-Kit as opposed to MP TM-Kit. By using a time management algorithm tailored to take advantage of shared memory SHM DRTI is able to achieve higher performance. However, when run on eight processors SHM DRTI has similar performance to MP DRTI.

9.9 Communication Architectures

Symmetric Multi-Processors (SMPs) have become affordable and widely available. They offer properties that can be exploited in order to speed real-world applications. SMPs offer mechanism for processors to share memory. This shared memory can be used to efficiently communicate between processors, processes, or threads. A distributed application running on a network of SMPs could use shared memory for communication on a particular SMP avoiding the cost of more expensive communication mechanisms such as TCP.

This below extend the above work and investigates software communication architectures for networked SMPs. The goal is to determine which architecture would give the best performance and most desirable properties. The overall goal of the research described in the chapter is to federate Georgia Tech Time Warp (GTW) simulations using RTI-Kit. As described previously, RTI-Kit provides the tools needed to build High Level Architecture (HLA) Run Time Infrastructures (RTIS).

9.9.1 Threads

Currently federates have a single thread of control. In order for communication to occur the federate must make a procedure call to allow the RTI to perform these services. This is done through the *tick* service. However, this could lead to added complexity in the federate. The federate must be designed so that enough time is given to the RTI to allow it to perform necessary communication services. For example, in order for a Lowest Bound Time Stamp (LBTS) computation to complete each federate must report its minimum time. This computation can be slowed down by have just one federate that does not allocate enough time to the RTI by calling *tick* an appropriate number of times. Also, if *tick* is not called often enough message will build up at the receiver side slowing overall communication between federates.

To free the federate from the having to call *tick* to allow communication services to be performed one could designate a thread to perform all communication for the federate. This communication thread would be responsible for pulling messages off the network, delivering the message to the federate, and other low level functions that do not require the intervention of the federate. The operating system scheduler would ensure that the communication thread be scheduled to run on a regular basis avoiding a buildup of messages at the receiver side.

9.9.2 Interrupts vs. Polling

In multi-threaded architectures there is the issue of how to notify the federate that a message has been received. Currently, the federate learns of messages when it calls *tick*. The *tick* call causes messages to be delivered to the federate through call backs that are implemented by the federate. There are two methods that the communication thread can notify the federate thread that a message has been received.

The communication thread can interrupt the federate and force the federate to deal with the message immediately. There are two problems with this approach. First, the federate can be interrupted at any point during its execution. In the current implementations messages will only be delivered to the federate when the federate calls *tick*. This means that messages will be delivered at known points during the federate's execution. This property is lost if the communication thread interrupts the federate when messages arrive. Second, if there is a constant stream of messages arriving at the federate it will be significantly slowed down. Each message that arrives and interrupts the federate prevents the federate from doing its own computation.

The alternative to interrupts is polling. Instead of the communication thread interrupting the federate thread when messages arrive the communication thread places the message in a buffer in shared memory. Periodically the federate checks the buffer for messages. In this case there is still a need for the federate to call *tick*, but in this case *tick* is only responsible for delivering messages to the federate and not giving time for low level communication services to be performed. With polling we retain the properties that time is automatically given to the communication thread to perform low level communication services, and messages are delivered only when the federate calls *tick*.

9.9.3 Four Communication Architectures

Next is a description of four communication architectures and a brief list of advantages and disadvantages for each architecture.

Architecture 1: No Separate Thread for Communication

Federate and communication services run within one process. The federate is required to call *tick* to give time for communication services to be performed.

Advantages: This architecture is simple. This is how federates are currently implemented, with one thread of control and calling *tick* to perform communication services.

Disadvantages: The biggest disadvantage is the federate is required to call *tick* to perform communication services. If *tick* is not called in a timely manner messages can accumulate at the receiver side. This can lead to unpredictable latency due to the fact that latencies are now dependent on when *tick* is called. In particular real time systems are dependent on predictable latency and extra care and complexity (in the application) would be require to obtain predictable latency with such a communication architecture.

Architecture 2: Multiple Communication Threads (One per Federate)

Each federate has a separate communication thread dedicated to performing the low level communication services.

Advantages: By having a separate thread performing the low level communication services there is no longer a dependence on the federate to call tick to allow communication services to be executed.

Disadvantages: The most significant overhead incurred by have a separate thread handling the communication is the extra hop (typically an extra memory copy) needed to get message from the federate thread to the communication thread and visa versa. Also, there is some overhead incurred by the operating system to manage the extra threads.

Architecture 3: Single Communication Thread (One per SMP)

Each SMP has one communication thread dedicated to performing the low level communication services for all federates running on that SMP.

Advantages: This architecture has the same advantages as the previous architecture. Now with only one extra thread per SMP there is less overhead incurred by the operating system to manage the extra threads.

Disadvantages: As with the previous architecture there is the extra hop needed within the SMP to send and receive.

Architecture 4: Dedicated Processor to Communication

One processor in the SMP is dedicated to performing the low level communication services for federates running on that SMP.

Advantages: With a processor devoted to handling communication for the entire SMP there is no longer the issue of overhead incurred by the operating system having to manage additional threads. The communication thread runs on its own processor and is immune to being de-scheduled by the operating system. This allows it to always be available to perform communication services.

Disadvantages: The biggest disadvantage is a processor has to be devoted to communication. If there is little communication between the federates then the processing power of that processor is being wasted.

9.10 Performance

Micro-benchmarks were written to gather performance results for each of the architectures presented above. The benchmarks measured the time required for a message to be sent to each processor in a network of SMPs. The plot in Figure 9.9 shows the average time to pass a message between two processors.

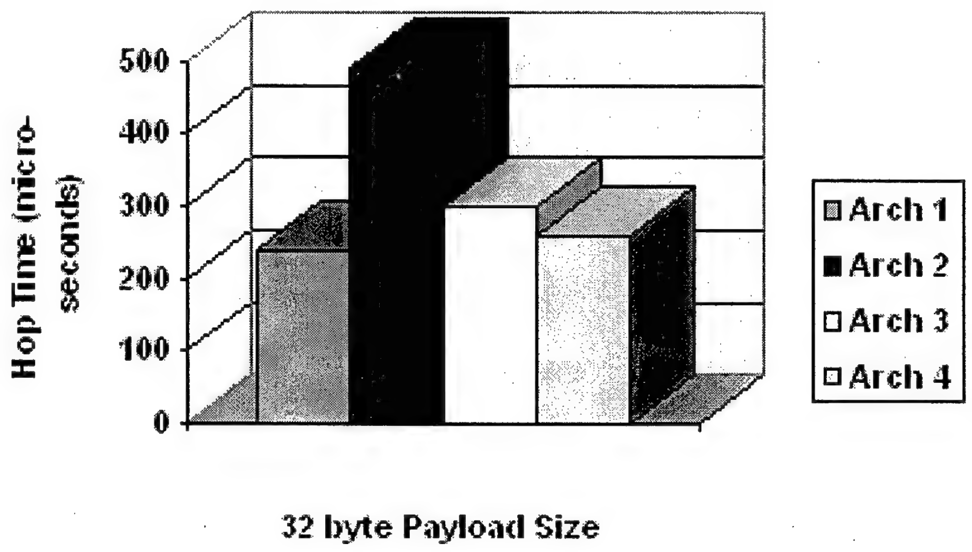


Figure 9.9: Average Hop Time Between Two Processors; Comparison of Four Communication Architectures.

From the benchmarks several things were learned. For Architectures 2, 3, and 4, the extra hop the message takes within the SMP when sending or receiving adds only about 15% overhead. Also, the extra cost of having a separate communication thread does not add a significant amount of overhead and is justified considering the benefits provided by having a separate thread for communication.

9.11 Conclusions and Future Work

The experiments here show that it is possible to implement RTIs on SMPs using time management algorithms specifically designed to take advantage of sequentially consistent shared memory. For up to eight processors, the performance of an RTI using the shared memory time management algorithm when compared to a message passing time management algorithm is typically about the same (for 8 processors) or better (for 2, 4, 6 processors).

We believe the decrease in performance for SHM TM-Kit on eight processors is due to contention for locks used in the critical sections of the shared memory algorithm. The more processors that participate in an LBTS computation increase the likelihood that there will be contention for the lock. Another cause for the degradation in performance as the number of processors is increased is traffic between the processors and memory. More processors mean more information will be transferred between the shared memory and the individual processors. Just as in a traditional network the more entities that try to send and receive information degrades overall system performance.

However, the use of SMPs to run HLA RTIs is promising. With more efficient implementations of time management algorithms (and efficient group communication algorithms,) SMPs will offer an efficient platform on which to execute RTIs. With this observation we also investigated four communication architecture and evaluated their performance. To continue this research direction we plan to incorporate architecture 2 and 3 for communication as they promised the best performance. Additional user control parameters will built in to RTI-Kit in order to allow an application writer to specify the communication architecture that promises the best communication paradigm for the given application. The will provide a platform enabling the federation of GTW simulations using RTI-Kit as the glue.

REFERENCES

ARTSY, Y. AND FINKEL, R. 1989. Designing a process migration facility: The charlotte experience. *IEEE Computer 22*, 9 (September), 47–53.

AVRIL, H. AND TROPPER, C. 1996. The dynamic load balancing of clustered time warp for logic simulation. In 10th *Workshop on Parallel and Distributed Simulation* (May 1996), 22–32.

BARAK, A., GUDAY, S., AND WHEELER, R. G. 1993. The MOSIX distributed operating system: Load balancing for Unix. In *Lecture Notes in Computer Science*, Number 672 (1993). Springer-Verlag.

BELLENOT, S. 1993. Performance of a riskfree time warp operating system. In 7th *Workshop on Parallel and Distributed Simulation*, Volume 23 (May 1993), 155–158.

BELLENOT, S. AND DUTY, L. 1995. XTracker, a graphical tool for parallel simulations. 191–194.

BERRY, O. AND JEFFERSON, D. 1985. Critical path analysis of distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation* (January 1985), 57–60.

BESTAVROS, A. 1994. Multi-version speculative concurrency control with delayed commit. In *Proceedings of the 1994 International Conference on Computers and their Applications* (1994).

BODEN, N. J., COHEN, D., FELDERMAN, R., KULAWIK, A. E., SETIZ, C. L., SEIZOVIC, J. N., AND SU, W.-K. 1995. Myrinet: A gigabit per second local area network. *IEEE-Mirco 15*, 1 (Feb.), 29–36.

BOUKERCHE, A. AND DAS, S. K. 1997. Dynamic load balancing strategies for conservative parallel simulations. In 11th *Workshop on Parallel and Distributed Simulation* (June 1997), 20–28. IEEE Computer Society Press.

BRINER, J., JR. 1991. Fast parallel simulation of digital systems. *23*, 1 (January), 71–77.

BROWN, R. 1988. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM 31*, 10 (October), 1220–1227.

BRYANT, R. E. 1977. Simulation of packet communication architecture computer systems. MIT-LCS-TR-188, Massachusetts Institute of Technology, Cambridge, Massachusetts.

BURDORF, C. AND MARTI, J. 1993. Load Balancing Strategies for Time Warp on Multi-User Workstations. *The Computer Journal 36*, 2, 168–176.

CAROTHERS, C. AND FUJIMOTO, R. M. 1996. Background execution of time warp programs. In 10th *Workshop on Parallel and Distributed Simulation* (May 1996), 12–19.

CAROTHERS, C., FUJIMOTO, R. M., AND ENGLAND, P. 1994. Effect of communication overheads on time warp performance. In 8th *Workshop on Parallel and Distributed Simulation* (July 1994), 118–125.

CAROTHERS, C., FUJIMOTO, R. M., AND LIN, Y.-B. 1995. A case study in simulating pcs networks using time warp. In 9th *Workshop on Parallel and Distributed Simulation* (June 1995), 87–94.

CAROTHERS, C., LIN, Y.-B., AND FUJIMOTO, R. M. 1995. A re-dial model for personal communications services networks. In *1995 Vehicular Technology Conference* (July 1995).

CHANCHIO, K. AND SUN, X.-H. 1996. Efficient process migration for parallel processing on non-dedicated networks of workstations. Technical Report 96-74 (December), ICASE.

CHANDY, K. M. AND MISRA, J. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering SE-5*, 5 (September), 440–452.

CHANDY, K. M. AND MISRA, J. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM 24*, 4 (April), 198–205.

COX., D. 1990. Personal communications – a viewpoint. *128*, 11.

DAS, S., FUJIMOTO, R., PANESAR, K., ALLISON, D., AND HYBINETTE, M. 1994. GTW: A Time Warp system for shared memory multiprocessors. In *1994 Winter Simulation Conference Proceedings* (December 1994), 1332–1339.

DAS, S. R. AND FUJIMOTO, R. M. 1994. An adaptive memory management protocol for Time Warp parallel simulation. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1994), 201–210.

DICKENS, P. M. AND REYNOLDS, JR., P. F. 1990. SRADS with local rollback. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 22 (January 1990), 161–164. SCS Simulation Series.

DMSO. 1998. Defense modeling and simulation office web site. `http://hla.dmso.mil`.

DONGARRA, J. J., OTTO, S. W., SNIR, M., AND WALKER, D. 1996. A message passing standard for mpp and workstations. *Communications of the ACM 39*, 7.

FRANKS, S., GOMES, F., UNGER, B., AND CLEARY, J. 1997. State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation* (1997), 72–79.

FUJIMOTO, R. AND HOARE, P. 1998. HLA RTI performance in high speed LAN environments. In *Proceedings of the Fall Simulation Interoperability Workshop* (September 1998).

FUJIMOTO, R. AND WEATHERLY, R. 1997. Time managment in the dod high level architecture. 60–67.

FUJIMOTO, R. M. 1989a. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation 6*, 3 (July), 211–239.

FUJIMOTO, R. M. 1989b. Time Warp on a shared memory multiprocessor. In *Proceedings of the 1989 International Conference on Parallel Processing*, Volume 3 (August 1989), 242–249.

FUJIMOTO, R. M. 1990a. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (October), 30–53.

FUJIMOTO, R. M. 1990b. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 22 (January 1990), 23–28. SCS Simulation Series.

FUJIMOTO, R. M. 1990c. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 22 (January 1990), 23–28. SCS Simulation Series.

FUJIMOTO, R. M. 1990d. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 22 (January 1990), 23–28. SCS Simulation Series.

FUJIMOTO, R. M. AND HYBINETTE, M. 1994. Computing global virtual time on shared-memory multiprocessors. Technical report (Aug.), College of Computing, Georgia Institute of Technology.

FUJIMOTO, R. M. AND HYBINETTE, M. 1997. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation 7*, 4 (October), 425–446.

FUJIMOTO, R. M., R.DAS, S., K. S, P., HYBINETTE, M., AND CAROTHERS, C. 1997. Georgia tech time warp programmer's manual for distributed network of workstations. Technical Report GIT-CC-97-18 (july), College of Computing, Georgia Institute of Technology.

GARCIA-MOLINA, H. AND SALEM, K. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (1987), 249–259.

GEIST, A., BEGUELIN, A., DONGARRA, J., JIAN, W., MANCHEK, R., AND SUNDERAM, V. 1993. PVM 3 user's guide and reference manual. Technical Report TM-12187 (May), Oak Ridge National Laboratory.

GEIST, G., HEATH, M., PEYTON, B., AND WORLEY, P. H. 1990. PICL: A portable instrumented communication library, C reference manual. Technical Report ORNL/TM-11130 (July), Mathematical Sciences Section, Oak Ridge National Laboratory.

GHOSH, K., FUJIMOTO, R. M., AND SCHWAN, K. 1993. Time warp simulation in time constrained systems. In 7th *Workshop on Parallel and Distributed Simulation*, Volume 23 (May 1993), 163–166. SCS Simulation Series.

GLASSERMAN, P., HEIDELBERGER, P., AND SHAHABUDDIN, P. 1996. Splitting for rare event simulation: Analysis of simple cases. In *Proceedings of the 1996 Winter Simulation Conference* (December 1996), 302–308.

GLAVACH, M. A. AND STURROCK, D. T. 1993. Introduction to siman/cinema. In *1993 Winter Simulation Conference Proceedings* (1993), 190–192.

GLAZER, D. W. AND TROPPER, C. 1993. On process migration and load balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems 3*, 4 (March), 318–327.

GLYNN, P. W. AND HEIDELBERGER, P. 1991. Analysis of parallel replicated simulations under a completion time constraint. *ACM Transactions on Modeling and Computer Simulation 1*, 1, 3–23.

GOLDBERG, A. P. 1992a. Virtual time synchronization of replicated processes. In 6th *Workshop on Parallel and Distributed Simulation*, Volume 24 (January 1992), 107–116. SCS Simulation Series.

GOLDBERG, A. P. 1992b. Virtual time synchronization of replicated processes. In 6th *Workshop on Parallel and Distributed Simulation*, Volume 24 (January 1992), 107–116. SCS Simulation Series.

GRAHAM, J. H., ELMAGHRABY, A. S., KARACHIWALA, I. S., AND SOLIMAN, H. 1996. A visual environment for distributed simulation systems. *Simulation Digest 25*, 3, 13–22.

GUPTA, A., AKYILDIZ, I. F., AND FUJIMOTO, R. M. 1991. Performance analysis of Time Warp with homogeneous processors and exponential task times. *19*, 1 (May), 101–110.

HAO, F., WILSON, K., FUJIMOTO, R. M., AND ZEGURA, E. 1996. Logical process size in parallel atm simulations. In *1996 Winter Simulation Conference Proceedings* (December 1996), 645–652.

HYBINETTE, M. AND FUJIMOTO, R. M. 1997. Cloning: a novel method for interactive parallel simulation. In *Proceedings of the 1997 Winter Simulation Conference* (December 1997), 444–451.

HYBINETTE, M. AND FUJIMOTO, R. M. 1998. Dynamic virtual logical processes. In 12th *Workshop on Parallel and Distributed Simulation* (May 1998), 100–107.

HYBINETTE, M. AND FUJIMOTO, R. M. 1999. Cloning parallel simulations: Interactive-Sim 0.9 – A programmer's manual and specifications. Technical report (August), College of Computing, Georgia Institute of Technology, Atlanta, GA. in progress.

JEFFERSON, D. R. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July), 404–425.

JEFFERSON, D. R. AND SOWIZRAL, H. 1982. Fast concurrent simulation using the Time Warp mechanism, part I: Local control. Technical Report N-1906-AF (December), RAND Corporation.

KANARICK, C. 1991. A technical overview and history of the SIMNET project. In *Advances in Parallel and Distributed Simulation*, Volume 23 (January 1991), 104–111. SCS Simulation Series.

KORTH, H., LEVY, E., AND SILBERSCHATZ, A. 1990. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases* (August 1990), 95–106.

KRAEMER, E. AND STASKO, J. T. 1993. The visualization of parallel systems: An overview. *Journal of Parallel and Distributed Computing 18*, 2 (June), 105–117.

LAMPORT, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9, 690–691.

LIPTON, R. J. AND MIZELL, D. W. 1990. Time Warp vs. Chandy-Misra: A worst-case comparison. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 22 (January 1990), 137–143. SCS Simulation Series.

LITZKOW, M. AND LIVNY, M. 1990. Experience with the condor distributed batch system. In *IEEE Workshop on Experimental Distributed Systems* (October 1990).

LUBACHEVSKY, B. D. 1989. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM 32*, 1 (Jan.), 111–123.

LUBACHEVSKY, B. D., SHWARTZ, A., AND WEISS, A. 1991. An analysis of rollback-based simulation. *ACM Transactions on Modeling and Computer Simulation 1*, 2 (April), 154–193.

MADISETTI, V. K., HARDAKER, D. A., AND FUJIMOTO, R. M. 1993. The mimdix operating system for parallel simulation and supercomputing. *Journal of Parallel and Distributed Computing 18*, 4 (August), 473–483.

MASCARENHAS, E., REGO, V., AND SANG, J. 1995. DISplay: A system for visual-interaction in distributed simulations. In *1995 Winter Simulation Conference Proceedings* (1995), 698–705.

MATTERN, F. 1993. Efficient distributed snapshots and global virtual time algorithms for non-fifo systems. *Journal of Parallel and Distributed Computing 18*, 4 (August), 423–434.

SILICON GRAPHICS, INC. 1998. SGI Origin 2000 technical specifications web site. `http://www.sgi.com/origin/2000_specs.html`.

THE MYRINET. 1998. A brief technical overview. `http://www.myri.com`.

MULLENDER, S. J., VAN ROSSUM, G., VAN RENESSE, R., AND VAN STAVEREN, H. 1990. Amoeba – a distributed operating system for the 1990s. *IEEE Computer 23*, 5 (May), 44–53.

NICOL, D. 1991. Performance bounds on parallel self-initiating discrete-event simulations. *ACM Transactions on Modeling and Computer Simulation 1*, 1 (January), 24–50.

NICOL, D. M. AND REYNOLDS, JR., P. F. 1984. Problem oriented protocol design. In *1984 Winter Simulation Conference Proceedings* (December 1984), 471–474.

OFFICE, T. P. 1995. Software product specification for the THAAD integrate system effectiveness simulation (TISES). Technical report (Sept.), User Documentation for TISES Software.

O'KEEF, R. M. 1987. What is visual interactive simulation? (And is there a methodology for doing it right?). In *1987 Winter Simulation Conference Proceedings* (1987), 461–464.

OUSTERHOUT, J. K., CHERENSON, A. R., DOUGLIS, F., NELSON, M. N., AND WELCH, B. B. 1988. The sprite network operating system. *IEEE Computer 21*, 2 (February), 23–36.

PAKIN, S., LARIA, M., BUCHANAN, M., HANE, K., GIANNINI, L., PRUSAKOVA, J., AND CHEIN, A. Fast message (FM) 2.0 user documentation. Technical report, Deptartment of Computer Science at the University of Illinois at Urbana Champaign.

PANESAR, K. AND FUJIMOTO, R. M. 1997. Adaptive flow control in time warp. In 11^{th} *Workshop on Parallel and Distributed Simulation* (June 1997).

PEACOCK, J. K., WONG, J. W., AND MANNING, E. G. 1979. Distributed simulation using a network of processors. *Computer Networks 3*, 1 (February), 44–56.

PRESLEY, M., EBLING, M., WIELAND, F., AND JEFFERSON, D. R. 1989. Benchmarking the Time Warp Operating System with a computer network simulation. *21*, 2 (March), 8–13.

REIHER, P. L. AND JEFFERSON, D. 1990. Dynamic load management in the Time Warp Operating System. *Transactions of the Society for Computer Simulation 7*, 2 (June), 91–120.

RICH, D. O. AND MICHELSEN, R. E. 1991. An assessment of the Modsim/TWOS parallel simulation environment. In *1991 Winter Simulation Conference Proceedings* (December 1991), 509–518.

RONNGREN, R. AND AYANI, R. 1997. Parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation 7*, 2 (April), 157–209.

ROSU, M.-C., SCHWAN, K., AND FUJIMOTO, R. M. 1997. Supporting parallel applications on clusters of workstations: the intelligent network interface approach. In *Proceedings of the 1997 IEEE High Performance Distributed Computing* (August 1997).

SCHLAGENHAFT, R., RUHWANDL, M., SPORRER, C., AND BAUER, H. 1995. Dynamic load balancing of a multi-cluster simulation on a network of workstations. In 9th *Workshop on Parallel and Distributed Simulation*, Volume 24 (1995), 175–180.

SCHMIDT, B. K. AND SUNDERAM, V. S. 1994. Empirical analysis of overheads in cluster environments. *Concurrency: Practice & Experience 6*, 1 (February), 1–33.

SCHNEIDER, F. 1990. Implementing fault-tolerance services using the state machine approach: A tutorial. *ACM Computer Surveys 22*, 4, 299–320.

SIMMONS, M., KOSKELA, R., AND BUCHER, I. 1989. Instrumentation for future parallel computing systems. *ACM Press Frontier Series*.

SLEATOR, D. D. AND TARJAN, R. E. 1985. Self-adjusting binary search trees. *Journal of the ACM 32*, 3 (July), 652–686.

SLEATOR, D. D. AND TARJAN, R. E. 1986. Self-adjusting heaps. *SIAM Journal on Computing 15*, 1 (Feb.), 52–59.

SOULE, L. AND GUPTA, A. 1991. An evaluation of the Chand-Misra-Byrant algorithm for digital logic simulation. *ACM Transactions on Modeling and Computer Simulation 1*, 4 (Oct.).

SRINIVASAN, S. AND REYNOLDS, JR., P. F. 1995. NPSI adaptive synchronization algorithms for PDES. In *1995 Winter Simulation Conference Proceedings* (December 1995), 658–665.

STEINMAN, J. 1991. Speedes: Synchronous parallel environment for emulation and discrete event simulation. In *Advances in Parallel and Distributed Simulation*, Volume 23 (January 1991), 95–103. SCS Simulation Series.

STEINMAN, J. 1992. Speedes: A multiple-synchronization environment for discrete-event simulation. *Internation Journal in Computer Simulation 2*, 3 (March), 251–286.

STEINMAN, J. S. 1993. Breathing time warp. In 7th *Workshop on Parallel and Distributed Simulation*, Volume 23 (May 1993), 109–118. SCS Simulation Series.

SUNDERAM, V. 1990. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience 2*, 4 (December), 315–339.

THEIMER, M. M., LANTZ, A., AND CHERITON, D. R. 1985. Preemptable remote execution facilities for the v-system. In *Proceedings of the* 10th *ACM Symposium on Operating System Principles* (1985).

THIEL, G. 1991. Locus operating system, a transparent system. *Computer Communications 14*, 6, 336–346.

THOMPSON, W. 1993. A tutorial for modeling with the WITNESS visual interactive simulator. In *1993 Winter Simulation Conference Proceedings* (1993), 228–232.

TOPOL, B., STASKO, J. T., AND SUNDERAM, V. PVaniM: A tool for visualization in network computing environments. *Concurrency: Practice & Experience*. to appear.

TOPOL, B., STASKO, J. T., AND SUNDERAM, V. 1997. The dual timestamping methodology for visualizing distributed applications. In *Proceedings of the IASTED International Conference on Parallel and Distributed Systems, (Euro-PDS '97)* (June 1997), 81–86.

VAIKILI, P. 1992. Massively parallel and distributed simulation of a class of discrete event systems: A different perspective. *ACM Transactions on Modeling and Computer Simulation 2*, 3, 214–238.

VON NEUMANN, J. 1956. *Probabilistic logics and the synthesis of reliable organism from unreliable components.* Princeton University Press.

WIELAND, F. 1997. The threshold of event simultaneity. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation* (1997), 56–59.

WIELAND, F., BLAIR, E., AND ZUKAS, T. 1995. Parallel discrete-event simulation (pdes): A case study in design, development, and performance using speedes. In 9th *Workshop on Parallel and Distributed Simulation*, Volume 24 (1995), 103–110.

WIELAND, F., HAWLEY, L., FEINBERG, A., DILORENTO, M., BLUME, L., REIHER, P., BECKMAN, B., HONTALAS, P., BELLENOT, S., AND JEFFERSON, D. R. 1989. Distributed combat simulation and Time Warp: The model and its performance. *21*, 2 (March), 14–20.

WILLEBEEK-LEMAIR, M. H. AND REEVE, A. P. 1993. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems 4*, 9 (Sept.), 979–993.

WILSON, A. L. AND WEATHERLY, R. M. 1994. The aggregate level simulation protocol: An evolving system. In *1994 Winter Simulation Conference Proceedings* (December 1994), 781–787.

WILSON, L. F. AND NICOL, D. M. 1996. Experiments in automated load balancing. In 10th *Workshop on Parallel and Distributed Simulation* (1996), 4–11. IEEE Computer Society Press.

XIAO, Z. AND GOMES, F. 1993. Benchmarking smtw with a ss7 performance model simulation. Technical report, University of Calgary. unpublished project report for CPSC 601.24 (Fall).

ZAYAS, E. 1987a. Attacking the process migration bottleneck. In *Proceedings of the* 11th *ACM Symposium on Operating System Principles* (1987), 13–24.

ZAYAS, E. 1987b. Attacking the process migration bottleneck. In *Proceedings of the* 11th *ACM Symposium on Operating System Principles* (1987), 13–24.

ZHU, W. AND GOSCINSKI, A. 1990. Process migration in rhodos. Technical Report CS90/9 (March), Department of Computer Science, University College, University of New South Wales.